Raster Vision Documentation

Release 0.13.0

Azavea

Oct 05, 2022

Contents

rastervision

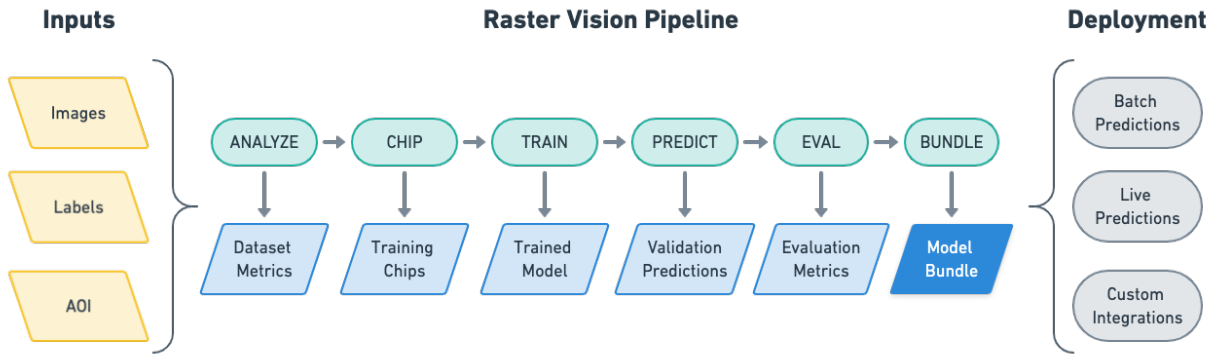
Raster Vision is an open source framework for Python developers building computer vision models on satellite, aerial, and other large imagery sets (including oblique drone imagery). There is built-in support for chip classification, object detection, and semantic segmentation using PyTorch.

Chip Classification

Object Detection

Semantic Segmentation

Raster Vision allows engineers to quickly and repeatably configure *pipelines* that go through core components of a machine learning workflow: analyzing training data, creating training chips, training models, creating predictions, evaluating models, and bundling the model files and configuration for easy deployment.



The input to a Raster Vision pipeline is a set of images and training data, optionally with Areas of Interest (AOIs) that describe where the images are labeled. The output of a Raster Vision pipeline is a model bundle that allows you to easily utilize models in various deployment scenarios.

The pipelines include running the following commands:

- **ANALYZE**: Gather dataset-level statistics and metrics for use in downstream processes.
- **CHIP**: Create training chips from a variety of image and label sources.

- **TRAIN**: Train a model using a "backend" such as PyTorch.
- **PREDICT**: Make predictions using trained models on validation and test data.
- **EVAL**: Derive evaluation metrics such as F1 score, precision and recall against the model's predictions on validation datasets.
- **BUNDLE**: Bundle the trained model and associated configuration into a *model bundle*, which can be deployed in batch processes, live servers, and other workflows.

Pipelines are configured using a compositional, programmatic approach that makes configuration easy to read, reuse, and maintain. Below, we show the `tiny_spacenet` example.

Listing 1: tiny_spacenet.py

```
# flake8: noqa

from rastervision.core.rv_pipeline import *
from rastervision.core.backend import *
from rastervision.core.data import *
from rastervision.pytorch_backend import *
from rastervision.pytorch_learner import *


def get_config(runner) -> SemanticSegmentationConfig:
    root_uri = '/opt/data/output/'
    base_uri = ('https://s3.amazonaws.com/azavea-research-public-data/'
                'raster-vision/examples/spacenet')

    train_image_uri = f'{base_uri}/RGB-PanSharpen_AOI_2_Vegas_img205.tif'
    train_label_uri = f'{base_uri}/buildings_AOI_2_Vegas_img205.geojson'
    val_image_uri = f'{base_uri}/RGB-PanSharpen_AOI_2_Vegas_img25.tif'
    val_label_uri = f'{base_uri}/buildings_AOI_2_Vegas_img25.geojson'

    channel_order = [0, 1, 2]
    class_config = ClassConfig(
        names=['building', 'background'], colors=['red', 'black'])

    def make_scene(scene_id: str, image_uri: str,
                   label_uri: str) -> SceneConfig:
        """
        - The GeoJSON does not have a class_id property for each geom,
          so it is inferred as 0 (ie. building) because the default_class_id
          is set to 0.
        - The labels are in the form of GeoJSON which needs to be rasterized
          to use as label for semantic segmentation, so we use a RasterizedSource.
        - The rasterizer set the background (as opposed to foreground) pixels
          to 1 because background_class_id is set to 1.
        """
        raster_source = RasterioSourceConfig(
            uris=[image_uri], channel_order=channel_order)
        vector_source = GeoJSONVectorSourceConfig(
            uri=label_uri, default_class_id=0, ignore_crs_field=True)
        label_source = SemanticSegmentationLabelSourceConfig(
            raster_source=RasterizedSourceConfig(
                vector_source=vector_source,
                rasterizer_config=RasterizerConfig(background_class_id=1)))
        return SceneConfig(
            id=scene_id,
```

(continues on next page)

(continued from previous page)

```
            raster_source=raster_source,
            label_source=label_source)

    scene_dataset = DatasetConfig(
        class_config=class_config,
        train_scenes=[
            make_scene('scene_205', train_image_uri, train_label_uri)
        ],
        validation_scenes=[
            make_scene('scene_25', val_image_uri, val_label_uri)
        ])

    # Use the PyTorch backend for the SemanticSegmentation pipeline.
    chip_sz = 300

    backend = PyTorchSemanticSegmentationConfig(
        data=SemanticSegmentationGeoDataConfig(
            scene_dataset=scene_dataset,
            window_opts=GeoDataWindowConfig(
                method=GeoDataWindowMethod.random,
                size=chip_sz,
                size_lims=(chip_sz, chip_sz + 1),
                max_windows=10)),
        model=SemanticSegmentationModelConfig(backbone=Backbone.resnet50),
        solver=SolverConfig(lr=1e-4, num_epochs=1, batch_sz=2))

    return SemanticSegmentationConfig(
        root_uri=root_uri,
        dataset=scene_dataset,
        backend=backend,
        train_chip_sz=chip_sz,
        predict_chip_sz=chip_sz)
```

Raster Vision uses a `unittest`-like method for executing pipelines. For instance, if the above was defined in *tiny_spacenet.py*, with the proper setup you could run the experiment on AWS Batch by running:

```
> rastervision run batch tiny_spacenet.py
```

See the *Quickstart* for a more complete description of using this example.

This part of the documentation guides you through all of the library's usage patterns.

CHAPTER 1

Why Raster Vision?

1.1 Why another deep learning library?

Most machine learning libraries implement the core functionality needed to build and train models, but leave the "plumbing" to users to figure out. This plumbing is the work of implementing a repeatable, configurable workflow that creates training data, trains models, makes predictions, and computes evaluations, and runs locally and in the cloud. Not giving this work the engineering effort it deserves often results in a bunch of hacky, one-off scripts that are not reusable.

In addition, most machine learning libraries cannot work out-of-the-box with massive, geospatial imagery. This is because of the format of the data (eg. GeoTIFF and GeoJSON), the massive size of each scene (eg. 10,000 x 10,000 pixels), the use of map coordinates (eg. latitude and longitude), the use of more than three channels (eg. infrared), patches of missing data (eg. NODATA), and the need to focus on irregularly-shaped AOIs (areas of interest) within larger images.

1.2 What are the benefits of using Raster Vision?

- Programmatically configure pipelines in a concise, modifiable, and reusable way, using abstractions such as *pipelines*, *backends*, *datasets*, and *scenes*.
- Let the framework handle the challenges and idiosyncrasies of doing machine learning on massive, geospatial imagery.
- Run pipelines and individual commands from the command line that execute in parallel, locally or on AWS Batch.
- Read files from HTTP, S3, the local filesystem, or anywhere with the pluggable *File Systems* architecture.
- Make predictions and build inference pipelines using a single "model bundle" which includes the trained model and associated metadata.
- *Customize* Raster Vision using the *plugins* architecture.

1.3 Who is Raster Vision for?

- Developers **new to deep learning** who want to get spun up on applying deep learning to imagery quickly or who want to leverage existing deep learning libraries like PyTorch for their projects simply.
- People who are **already applying deep learning** to problems and want to make their processes more robust, faster and scalable.
- Machine Learning engineers who are **developing new deep learning capabilities** they want to plug into a framework that allows them to focus on the interesting problems.
- **Teams building models collaboratively** that are in need of ways to share model configurations and create repeatable results in a consistent and maintainable way.

CHAPTER 2

Quickstart

In this Quickstart, we'll train a semantic segmentation model on SpaceNet data. Don't get too excited - we'll only be training for a very short time on a very small training set! So the model that is created here will be pretty much worthless. But! These steps will show how Raster Vision pipelines are set up and run, so when you are ready to run against a lot of training data for a longer time on a GPU, you'll know what you have to do. Also, we'll show how to make predictions on the data using a model we've already trained on GPUs to show what you can expect to get out of Raster Vision.

For the Quickstart we are going to be using one of the published *Docker Images* as it has an environment with all necessary dependencies already installed.

See also:

It is also possible to install Raster Vision using `pip`, but it can be time-consuming and error-prone to install all the necessary dependencies. See *Installing via pip* for more details.

Note: This Quickstart requires a Docker installation. We have tested this with Docker 19, although you may be able to use a lower version. See Get Started with Docker for installation instructions.

You'll need to choose two directories, one for keeping your configuration source file and another for holding experiment output. Make sure these directories exist:

```
> export RV_QUICKSTART_CODE_DIR=`pwd`/code
> export RV_QUICKSTART_OUT_DIR=`pwd`/output
> mkdir -p ${RV_QUICKSTART_CODE_DIR} ${RV_QUICKSTART_OUT_DIR}
```

Now we can run a console in the the Docker container by doing

```
> docker run --rm -it \
    -v ${RV_QUICKSTART_CODE_DIR}:/opt/src/code  \
    -v ${RV_QUICKSTART_OUT_DIR}:/opt/data/output \
    quay.io/azavea/raster-vision:pytorch-0.13 /bin/bash
```

See also:

See *Docker Images* for more information about setting up Raster Vision with Docker images.

2.1 The Data

2.2 Configuring a semantic segmentation pipeline

Create a Python file in the `${RV_QUICKSTART_CODE_DIR}` named `tiny_spacenet.py`. Inside, you're going to write a function called `get_config` that returns a `SemanticSegmentationConfig` object. This object's type is a subclass of `PipelineConfig`, and configures a semantic segmentation pipeline which analyzes the imagery, creates training chips, trains a model, makes predictions on validation scenes, evaluates the predictions, and saves a model bundle.

Listing 1: tiny_spacenet.py

```
# flake8: noqa

from rastervision.core.rv_pipeline import *
from rastervision.core.backend import *
from rastervision.core.data import *
from rastervision.pytorch_backend import *
from rastervision.pytorch_learner import *


def get_config(runner) -> SemanticSegmentationConfig:
    root_uri = '/opt/data/output/'
    base_uri = ('https://s3.amazonaws.com/azavea-research-public-data/'
                'raster-vision/examples/spacenet')

    train_image_uri = f'{base_uri}/RGB-PanSharpen_AOI_2_Vegas_img205.tif'
    train_label_uri = f'{base_uri}/buildings_AOI_2_Vegas_img205.geojson'
    val_image_uri = f'{base_uri}/RGB-PanSharpen_AOI_2_Vegas_img25.tif'
    val_label_uri = f'{base_uri}/buildings_AOI_2_Vegas_img25.geojson'

    channel_order = [0, 1, 2]
    class_config = ClassConfig(
        names=['building', 'background'], colors=['red', 'black'])

    def make_scene(scene_id: str, image_uri: str,
                   label_uri: str) -> SceneConfig:
        """
        - The GeoJSON does not have a class_id property for each geom,
          so it is inferred as 0 (ie. building) because the default_class_id
          is set to 0.
        - The labels are in the form of GeoJSON which needs to be rasterized
          to use as label for semantic segmentation, so we use a RasterizedSource.
        - The rasterizer set the background (as opposed to foreground) pixels
          to 1 because background_class_id is set to 1.
        """
        raster_source = RasterioSourceConfig(
            uris=[image_uri], channel_order=channel_order)
        vector_source = GeoJSONVectorSourceConfig(
            uri=label_uri, default_class_id=0, ignore_crs_field=True)
        label_source = SemanticSegmentationLabelSourceConfig(
            raster_source=RasterizedSourceConfig(
```

(continues on next page)

(continued from previous page)

```
                vector_source=vector_source,
                rasterizer_config=RasterizerConfig(background_class_id=1)))
        return SceneConfig(
            id=scene_id,
            raster_source=raster_source,
            label_source=label_source)

    scene_dataset = DatasetConfig(
        class_config=class_config,
        train_scenes=[
            make_scene('scene_205', train_image_uri, train_label_uri)
        ],
        validation_scenes=[
            make_scene('scene_25', val_image_uri, val_label_uri)
        ])

    # Use the PyTorch backend for the SemanticSegmentation pipeline.
    chip_sz = 300

    backend = PyTorchSemanticSegmentationConfig(
        data=SemanticSegmentationGeoDataConfig(
            scene_dataset=scene_dataset,
            window_opts=GeoDataWindowConfig(
                method=GeoDataWindowMethod.random,
                size=chip_sz,
                size_lims=(chip_sz, chip_sz + 1),
                max_windows=10)),
        model=SemanticSegmentationModelConfig(backbone=Backbone.resnet50),
        solver=SolverConfig(lr=1e-4, num_epochs=1, batch_sz=2))

    return SemanticSegmentationConfig(
        root_uri=root_uri,
        dataset=scene_dataset,
        backend=backend,
        train_chip_sz=chip_sz,
        predict_chip_sz=chip_sz)
```

2.3 Running the pipeline

We can now run the pipeline by invoking the following command inside the container.

```
> rastervision run local code/tiny_spacenet.py
```

2.4 Seeing Results

If you go to `${RV_QUICKSTART_OUT_DIR}` you should see a directory structure like this.

Note: This uses the `tree` command which you may need to install first.

```
> tree -L 3
.
├── analyze
│   └── stats.json
├── bundle
│   └── model-bundle.zip
├── chip
│   └── 3113ff8c-5c49-4d3c-8ca3-44d412968108.zip
├── eval
│   └── eval.json
├── pipeline-config.json
├── predict
│   └── scene_25.tif
└── train
    ├── dataloaders
    │   ├── test.png
    │   ├── train.png
    │   └── valid.png
    ├── last-model.pth
    ├── learner-config.json
    ├── log.csv
    ├── model-bundle.zip
    ├── tb-logs
    │   └── events.out.tfevents.1585513048.086fdd4c5530.214.0
    ├── test_metrics.json
    └── test_preds.png
```

The root directory contains a serialized JSON version of the configuration at `pipeline-config.json`, and each subdirectory with a command name contains output for that command. You can see test predictions on a batch of data in `train/test_preds.png`, and evaluation metrics in `eval/eval.json`, but don't get too excited! We trained a model for 1 epoch on a tiny dataset, and the model is likely making random predictions at this point. We would need to train on a lot more data for a lot longer for the model to become good at this task.

2.5 Model Bundles

To immediately use Raster Vision with a fully trained model, one can make use of the pretrained models in our *Model Zoo*. However, be warned that these models probably won't work well on imagery taken in a different city, with a different ground sampling distance, or different sensor.

For example, to use a DeepLab/Resnet50 model that has been trained to do building segmentation on Las Vegas, one can type:

```
> rastervision predict https://s3.amazonaws.com/azavea-research-public-data/raster-
↪vision/examples/model-zoo-0.13/spacenet-vegas-buildings-ss/model-bundle.zip https://
↪s3.amazonaws.com/azavea-research-public-data/raster-vision/examples/model-zoo-0.13/
↪spacenet-vegas-buildings-ss/sample-predictions/sample-img-spacenet-vegas-buildings-
↪ss.tif prediction
```

This will make predictions on the image `1929.tif` using the provided model bundle, and will produce a file called `predictions.tif`. These files are in GeoTiff format, and you will need a GIS viewer such as QGIS to open them correctly on your device. Notice that the prediction package and the input raster are transparently downloaded via HTTP. The input image (false color) and predictions are reproduced below.

See also:

You can read more about the *model bundle* concept and the *predict* CLI command in the documentation.

2.6 Next Steps

This is just a quick example of a Raster Vision pipeline. For several complete example of how to train models on open datasets (including SpaceNet), optionally using GPUs on AWS Batch, see the *Examples*.

CHAPTER 3

Setup

3.1 Docker Images

Using the Docker images published for Raster Vision makes it easy to use a fully set up environment. We have tested this with Docker 19, although you may be able to use a lower version.

The images we publish include plugins and dependencies for using Raster Vision with PyTorch, and AWS S3 and Batch. These are published to quay.io/azavea/raster-vision. To run the container for the latest release, run:

```
> docker run --rm -it quay.io/azavea/raster-vision:pytorch-0.13 /bin/bash
```

There are also images with the *-latest* suffix for the latest commits on the `master` branch. You'll likely need to mount volumes and expose ports to make this container fully useful; see the docker/run script for an example usage.

You can also base your own Dockerfiles off the Raster Vision image to use with your own codebase. See *Bootstrap new projects with a template* for more information.

3.1.1 Docker Scripts

There are several scripts under docker/ in the Raster Vision repo that make it easier to build the Docker images from scratch, and run the container in various ways. These are useful if you are experimenting with changes to the Raster Vision source code, or writing *plugins*.

After cloning the repo, you can build the Docker image using:

```
> docker/build
```

Before running the container, set an environment variable to a local directory in which to store data.

```
> export RASTER_VISION_DATA_DIR="/path/to/data"
```

To run a Bash console in the PyTorch Docker container use:

```
> docker/run
```

This will mount the `$RASTER_VISION_DATA_DIR` local directory to to `/opt/data/` inside the container.

This script also has options for forwarding AWS credentials, and running Jupyter notebooks which can be seen below.

```
> docker/run --help

Usage: run <options> <command>

Run a console in a Raster Vision Docker image locally.
By default, the raster-vision-pytorch image is used in the CPU runtime.

Environment variables:
RASTER_VISION_DATA_DIR (directory for storing data; mounted to /opt/data)
RASTER_VISION_NOTEBOOK_DIR (optional directory for Jupyter notebooks; mounted to /opt/
↪notebooks)
AWS_PROFILE (optional AWS profile)

Options:
--aws forwards AWS credentials (sets AWS_PROFILE env var and mounts ~/.aws to /root/.
↪aws)
--tensorboard maps port 6006
--name sets the name of the running container
--jupyter forwards port 8888, mounts RASTER_VISION_NOTEBOOK_DIR to /opt/notebooks,␣
↪and runs Jupyter
--docs runs the docs server and forwards port 8000
--debug forwards port 3000 for use with remote debugger
--gpu use nvidia runtime

All arguments after above options are passed to 'docker run'.
```

3.2 Installing via pip

Rather than running Raster Vision from inside a Docker container, you can directly install the library using `pip`. However, we recommend using the Docker images since it can be difficult to install some of the dependencies.

```
> pip install rastervision==0.13
```

Note: Raster Vision requires Python 3.6 or later. Use `pip3 install rastervision==0.13.0` if you have more than one version of Python installed.

You will also need various dependencies that are not pip-installable. For an example of setting these up, see the Dockerfile.

3.2.1 Install individual pip packages

Raster Vision is comprised of a required `rastervision.pipeline` package, and a number of optional plugin packages, as described in *Codebase Overview*. Each of these packages have their own dependencies, and can be installed individually. Running the following command:

```
> pip install rastervision==0.13
```

is equivalent to running the following sequence of commands:

```
> pip install rastervision_pipeline==0.13
> pip install rastervision_aws_s3==0.13
> pip install rastervision_aws_batch==0.13
> pip install rastervision_core==0.13
> pip install rastervision_pytorch_learner==0.13
> pip install rastervision_pytorch_backend==0.13
> pip install rastervision_gdal_vsi==0.13
```

3.2.2 Troubleshooting macOS Installation

If you encounter problems running `pip install rastervision==0.13` on macOS, you may have to manually install Cython and pyproj.

To circumvent a problem installing pyproj with Python 3.7, you may also have to install that library using `git+https`:

```
> pip install cython
> pip install git+https://github.com/jswhit/pyproj.
↪git@e56e879438f0a1688b89b33228ebda0f0d885c19
> pip install rastervision==0.13.0
```

3.3 Raster Vision Configuration

Raster Vision is configured via the everett library, and will look for configuration in the following locations, in this order:

- Environment Variables
- A `.env` file in the working directory that holds environment variables.
- Raster Vision INI configuration files

By default, Raster Vision looks for a configuration file named `default` in the `${HOME}/.rastervision` folder.

3.3.1 Profiles

Profiles allow you to specify profile names from the command line or environment variables to determine which settings to use. The configuration file used will be named the same as the profile: if you had two profiles (the `default` and one named `myprofile`), your `${HOME}/.rastervision` would look like this:

```
> ls ~/.rastervision
default    myprofile
```

Use the `rastervision --profile` option in the *Command Line Interface* to set the profile.

3.3.2 Configuration File Sections

AWS_S3

```
[AWS_S3]
requester_pays = False
```

- `requester_pays` - Set to True if you would like to allow using requester pays S3 buckets. The default value is False.

Other

Other configurations are documented elsewhere:

- *AWS Batch Configuration*

3.3.3 Environment Variables

Any profile file option can also be stated in the environment. Just prepend the section name to the setting name, e.g. `export AWS_S3_REQUESTER_PAYS="False"`.

In addition to those environment variables that match the INI file values, there are the following environment variable options:

- `TMPDIR` - Setting this environment variable will cause all temporary directories to be created inside this folder. This is useful, for example, when you have a Docker container setup that mounts large network storage into a specific directory inside the Docker container. The tmp_dir can also be set on *Command Line Interface* as a root option.
- `RV_CONFIG` - Optional path to the specific Raster Vision Configuration file. These configurations will override configurations that exist in configurations files in the default locations, but will not cause those configurations to be ignored.
- `RV_CONFIG_DIR` - Optional path to the directory that contains Raster Vision configuration. Defaults to `${HOME}/.rastervision`

3.4 Running on a machine with GPUs

If you would like to run Raster Vision in a Docker container with GPUs - e.g. if you have your own GPU machine or you spun up a GPU-enabled machine on a cloud provider like a p3.2xlarge on AWS - you'll need to check some things so that the Docker container can utilize the GPUs.

Here are some severely outdated, but potentially still useful instructions written by a community member on setting up an AWS account and a GPU-enabled EC2 instance to run Raster Vision.

3.4.1 Install nvidia-docker

You'll need to install the nvidia-docker runtime on your system. Follow their Quickstart and installation instructions. Make sure that your GPU is supported by NVIDIA Docker - if not you might need to find another way to have your Docker container communicate with the GPU. If you figure out how to support more GPUs, please let us know so we can add the steps to this documentation!

3.4.2 Use the nvidia-docker runtime

When running your Docker container, be sure to include the `--runtime=nvidia` option, e.g.

```
> docker run --runtime=nvidia --rm -it quay.io/azavea/raster-vision:pytorch-0.13 /bin/
↪bash
```

or use the `--gpu` option with the `docker/run` script.

3.4.3 Ensure your setup sees the GPUS

We recommend you ensure that the GPUs are actually enabled. If you don't, you may run a training job that you think is using the GPU and isn't, and runs very slowly.

One way to check this is to make sure PyTorch can see the GPU(s). To do this, open up a `python` console and run the following:

```
import torch
torch.cuda.is_available()
torch.cuda.get_device_name(0)
```

This should print out something like:

```
True
Tesla K80
```

If you have nvidia-smi installed, you can also use this command to inspect GPU utilization while the training job is running:

```
> watch -d -n 0.5 nvidia-smi
```

3.5 Setting up AWS Batch

To run Raster Vision using AWS Batch, you'll need to setup your AWS account with a specific set of Batch resources, which you can do using *Setup AWS Batch using CloudFormation*.

3.5.1 AWS Batch Configuration

After creating the resources on AWS, set the following configuration in your *Raster Vision Configuration*. Check the AWS Batch console to see the names of the resources that were created, as they vary depending on how CloudFormation was configured.

```
[BATCH]
gpu_job_queue=RasterVisionGpuJobQueue
gpu_job_def=RasterVisionHostedPyTorchGpuJobDefinition
cpu_job_queue=RasterVisionCpuJobQueue
cpu_job_def=RasterVisionHostedPyTorchCpuJobDefinition
attempts=5
```

- `gpu_job_queue` - job queue for GPU jobs
- `gpu_job_def` - job definition that defines the GPU Batch jobs
- `cpu_job_queue` - job queue for CPU-only jobs

- `cpu_job_def` - job definition that defines the CPU-only Batch jobs
- `attempts` - Optional number of attempts to retry failed jobs. It is good to set this to > 1 since Batch often kills jobs for no apparent reason.

See also:

For more information about how Raster Vision uses AWS Batch, see the section: *Running remotely*.

CHAPTER 4

Command Line Interface

The Raster Vision command line utility, `rastervision`, is installed with a `pip install` of `rastervision`, which is installed by default in the *Docker Images*. It has a main command, with some top level options, and several subcommands.

```
> rastervision --help

 Usage: rastervision [OPTIONS] COMMAND [ARGS]...

 The main click command.

 Sets the profile, verbosity, and tmp_dir in RVConfig.

 Options:
 -p, --profile TEXT  Sets the configuration profile name to use.
 -v, --verbose       Increment the verbosity level.
 --tmpdir TEXT       Root of temporary directories to use.
 --help              Show this message and exit.

 Commands:
 predict      Use a model bundle to predict on new images.
 run          Run sequence of commands within pipeline(s).
 run_command  Run an individual command within a pipeline.
```

4.1 Subcommands

4.1.1 run

Run is the main interface into running pipelines.

```
> rastervision run --help

```

(continues on next page)

(continued from previous page)

```
Usage: rastervision run [OPTIONS] RUNNER CFG_MODULE [COMMANDS]...

Run COMMANDS within pipelines in CFG_MODULE using RUNNER.

RUNNER: name of the Runner to use

CFG_MODULE: the module with `get_configs` function that returns
PipelineConfigs. This can either be a Python module path or a local path
to a .py file.

COMMANDS: space separated sequence of commands to run within pipeline. The
order in which to run them is based on the Pipeline.commands attribute. If
this is omitted, all commands will be run.

Options:
-a, --arg            KEY VALUE   Arguments to pass to get_config function
-s, --splits         INTEGER     Number of processes to run in parallel for splittable
                                 commands
--pipeline-run-name  TEXT        The name for this run of the pipeline.
--help                           Show this message and exit.
```

Some specific parameters to call out:

–splits

Use `-s N` or `--splits N`, where `N` is the number of splits to create, to parallelize commands that can be split into parallelizable chunks. See *Running Commands in Parallel* for more information.

4.1.2 run_command

The `run_command` is used to run a specific command from a serialized `PipelineConfig` JSON file. This is likely only interesting to people writing *custom runners*.

```
> rastervision run_command --help

Usage: rastervision run_command [OPTIONS] CFG_JSON_URI COMMAND

Run a single COMMAND using a serialized PipelineConfig in CFG_JSON_URI.

Options:
--split-ind INTEGER   The process index of a split command
--num-splits INTEGER  The number of processes to use for running splittable
                        commands
--runner TEXT         Name of runner to use
--help                Show this message and exit.
```

4.1.3 predict

Use `predict` to make predictions on new imagery given a *model bundle*.

```
> rastervision predict --help

```

(continues on next page)

(continued from previous page)

```
Usage: rastervision predict [OPTIONS] MODEL_BUNDLE IMAGE_URI LABEL_URI

Make predictions on the images at IMAGE_URI using MODEL_BUNDLE and store
the prediction output at LABEL_URI.

Options:
--vector-label-uri TEXT  URI to save vectorized labels for semantic
                         segmentation model bundles that support it
-a, --update-stats       Run an analysis on this individual image, as
                         opposed to using any analysis like statistics that
                         exist in the prediction package
--channel-order TEXT     List of indices comprising channel_order. Example:
                         2 1 0
--help                   Show this message and exit.
```

CHAPTER 5

Pipelines and Commands

In addition to providing abstract pipeline functionality, Raster Vision provides a set of concrete pipelines for deep learning on remote sensing imagery including `ChipClassification`, `SemanticSegmentation`, and `ObjectDetection`. These pipelines all derive from `RVPipeline`, and are provided by the `rastervision.core` package. It's possible to customize these pipelines as well as create new ones from scratch, which is discussed in *Customizing Raster Vision*.

Chip Classification

Object Detection

Semantic Segmentation

5.1 Chip Classification

In chip classification, the goal is to divide the scene up into a grid of cells and classify each cell. This task is good for getting a rough idea of where certain objects are located, or where indiscrete "stuff" (such as grass) is located. It requires relatively low labeling effort, but also produces spatially coarse predictions. In our experience, this task trains the fastest, and is easiest to configure to get "decent" results.

5.2 Object Detection

In object detection, the goal is to predict a bounding box and a class around each object of interest. This task requires higher labeling effort than chip classification, but has the ability to localize and individuate objects. Object detection models require more time to train and also struggle with objects that are very close together. In theory, it is straightforward to use object detection for counting objects.

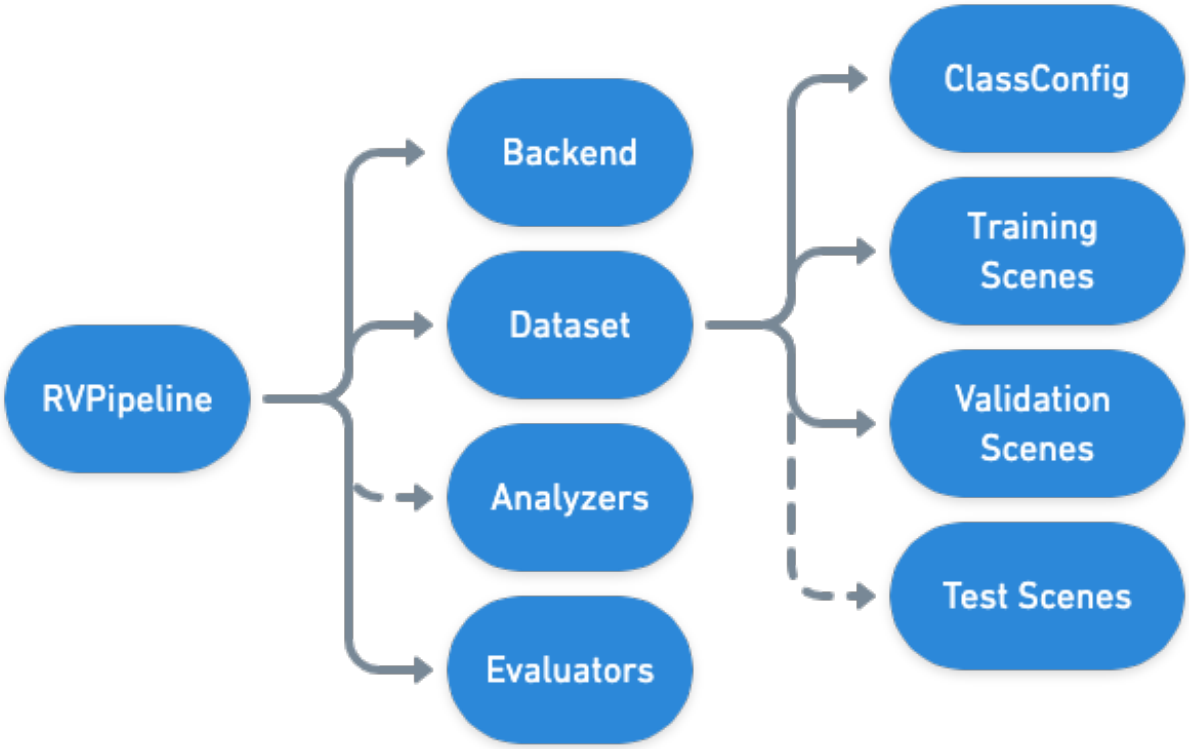
5.3 Semantic Segmentation

In semantic segmentation, the goal is to predict the class of each pixel in a scene. This task requires the highest labeling effort, but also provides the most spatially precise predictions. Like object detection, these models take longer to train than chip classification models.

5.4 Configuring RVPipelines

Each (subclass of) `RVPipeline` is configured by returning an instance of (a subclass of) `RVPipelineConfig` from a `get_config()` function in a Python module. It's also possible to return a list of `RVPipelineConfigs` from `get_configs()`, which will be executed in parallel.

Each `RVPipelineConfig` object specifies the details about how the commands within the pipeline will execute (ie. which files, what methods, what hyperparameters, etc.). In contrast, the *pipeline runner*, which actually executes the commands in the pipeline, is specified as an argument to the *Command Line Interface*. The following diagram shows the hierarchy of the high level components comprising an `RVPipeline`:



In the `tiny_spacenet.py` example, the `SemanticSegmentationConfig` is the last thing constructed and returned from the `get_config` function.

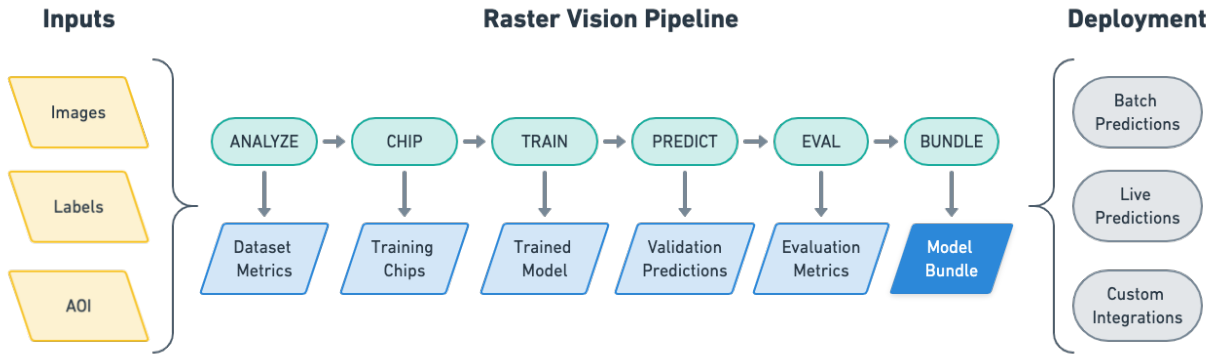
```
chip_sz = 300

backend = PyTorchSemanticSegmentationConfig(
    data=SemanticSegmentationGeoDataConfig(
        scene_dataset=scene_dataset,
        window_opts=GeoDataWindowConfig(
            method=GeoDataWindowMethod.random,
            size=chip_sz,
            size_lims=(chip_sz, chip_sz + 1),
            max_windows=10)),
    model=SemanticSegmentationModelConfig(backbone=Backbone.resnet50),
    solver=SolverConfig(lr=1e-4, num_epochs=1, batch_sz=2))

return SemanticSegmentationConfig(
    root_uri=root_uri,
    dataset=scene_dataset,
    backend=backend,
    train_chip_sz=chip_sz,
    predict_chip_sz=chip_sz)
```

See also:

The *ChipClassificationConfig*, *ObjectDetectionConfig*, and *SemanticSegmentationConfig* API docs have more information on configuring pipelines.

5.5 Commands

The `RVPipelines` provide a sequence of commands, which are described below.



5.5.1 ANALYZE

The ANALYZE command is used to analyze scenes that are part of an experiment and produce some output that can be consumed by later commands. Geospatial raster sources such as GeoTIFFs often contain 16- and 32-bit pixel color values, but many deep learning libraries expect 8-bit values. In order to perform this transformation, we need to know the distribution of pixel values. So one usage of the ANALYZE command is to compute statistics of the raster sources and save them to a JSON file which is later used by the StatsTransformer (one of the available *RasterTransformer*) to do the conversion.

5.5.2 CHIP

Scenes are comprised of large geospatial raster sources (e.g. GeoTIFFs) and geospatial label sources (e.g. GeoJSONs), but models can only consume small images (i.e. chips) and labels in pixel based-coordinates. In addition, each *Backend* has its own dataset format. The CHIP command solves this problem by converting scenes into training chips and into a format the backend can use for training.

5.5.3 TRAIN

The TRAIN command is used to train a model using the dataset generated by the CHIP command. The command uses the *Backend* to run a training loop that saves the model and other artifacts each epoch. If the training command is interrupted, it will resume at the last epoch when restarted.

5.5.4 PREDICT

The PREDICT command makes predictions for a set of scenes using a model produced by the TRAIN command. To do this, a sliding window is used to feed small images into the model, and the predictions are transformed from image-centric, pixel-based coordinates into scene-centric, map-based coordinates.

5.5.5 EVAL

The EVAL command evaluates the quality of models by comparing the predictions generated by the PREDICT command to ground truth labels. A variety of metrics including F1, precision, and recall are computed for each class (as well as overall) and are written to a JSON file.

5.5.6 BUNDLE

The BUNDLE command generates a model bundle from the output of the previous commands which contains a model file plus associated configuration data. A model bundle can be used to make predictions on new imagery using the *predict* command.

5.6 Backend

The collection of `RVPipelines` use a "backend" abstraction inspired by Keras, which makes it easier to customize the code for building and training models (including using Raster Vision with arbitrary deep learning libraries). Each backend is a subclass of `Backend` and has methods for saving training chips, training models, and making predictions, and is configured with a `BackendConfig`.

The `rastervision.pytorch_backend` plugin provides backends that are thin wrappers around the `rastervision.pytorch_learner` package, which does most of the heavy lifting of building and training models using `torch` and `torchvision`. (Note that `rastervision.pytorch_learner` is decoupled from `rastervision.pytorch_backend` so that it can be used in conjunction with `rastervision.pipeline` to write arbitrary computer vision pipelines that have nothing to do with remote sensing.)

Here are the PyTorch backends:

- The `PyTorchChipClassification` backend trains classification models from torchvision.
- The `PyTorchObjectDetection` backend trains the Faster-RCNN model in torchvision.
- The `PyTorchSemanticSegmentation` backend trains the DeepLabV3 model in torchvision.

In our `tiny_spacenet.py` example, we configured the PyTorch semantic segmentation backend using:

```
backend = PyTorchSemanticSegmentationConfig(
    data=SemanticSegmentationGeoDataConfig(
        scene_dataset=scene_dataset,
        window_opts=GeoDataWindowConfig(
            method=GeoDataWindowMethod.random,
            size=chip_sz,
            size_lims=(chip_sz, chip_sz + 1),
            max_windows=10)),
    model=SemanticSegmentationModelConfig(backbone=Backbone.resnet50),
    solver=SolverConfig(lr=1e-4, num_epochs=1, batch_sz=2))
```

See also:

The *rastervision.pytorch_backend* and *rastervision.pytorch_learner* API docs have more information on configuring backends.

5.7 Dataset

A `Dataset` contains the training, validation, and test splits needed to train and evaluate a model. Each dataset split is a list of `Scenes`.

In our `tiny_spacenet.py` example, we configured the dataset with single scenes, though more often in real use cases you would use multiple scenes per split:

```
scene_dataset = DatasetConfig(
    class_config=class_config,
    train_scenes=[
        make_scene('scene_205', train_image_uri, train_label_uri)
    ],
    validation_scenes=[
        make_scene('scene_25', val_image_uri, val_label_uri)
    ])
```

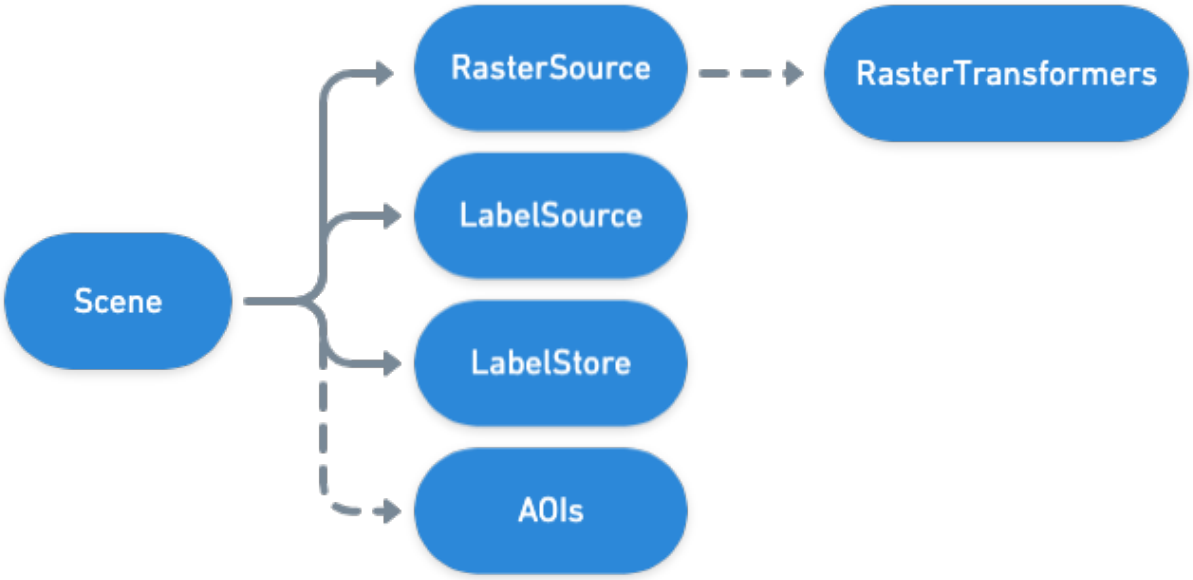
See also:

The *DatasetConfig* API docs.

5.8 Scene

A scene is composed of the following elements:

- *Imagery*: a *RasterSource* represents a large scene image, which can be made up of multiple sub-images or a single file.
- *Ground truth labels*: a *LabelSource* represents ground-truth task-specific labels.
- *Predicted labels*: a *LabelStore* determines how to store and retrieve the predictions from a scene.
- *AOIs* (Optional): An optional list of areas of interest that describes which sections of the scene imagery are exhaustively labeled. It is important to only create training chips from parts of the scenes that have been exhaustively labeled – in other words, that have no missing labels.



In our `tiny_spacenet.py` example, we configured the one training scene with a GeoTIFF URI and a GeoJSON URI.

```
def make_scene(scene_id: str, image_uri: str,
               label_uri: str) -> SceneConfig:
    """
    - The GeoJSON does not have a class_id property for each geom,
      so it is inferred as 0 (ie. building) because the default_class_id
      is set to 0.
    - The labels are in the form of GeoJSON which needs to be rasterized
      to use as label for semantic segmentation, so we use a RasterizedSource.
    - The rasterizer set the background (as opposed to foreground) pixels
      to 1 because background_class_id is set to 1.
    """
    raster_source = RasterioSourceConfig(
        uris=[image_uri], channel_order=channel_order)
    vector_source = GeoJSONVectorSourceConfig(
        uri=label_uri, default_class_id=0, ignore_crs_field=True)
    label_source = SemanticSegmentationLabelSourceConfig(
        raster_source=RasterizedSourceConfig(
            vector_source=vector_source,
            rasterizer_config=RasterizerConfig(background_class_id=1)))
    return SceneConfig(
        id=scene_id,
        raster_source=raster_source,
        label_source=label_source)
```

See also:

The *SceneConfig* API docs.

5.9 RasterSource

A `RasterSource` represents a source of raster data for a scene, and has subclasses for various data sources. They are used to retrieve small windows of raster data from larger scenes. You can also set a subset of channels (i.e. bands) that you want to use and their order. For example, satellite imagery often contains more than three channels, but pretrained models trained on datasets like Imagenet only support three (RGB) input channels. In order to cope with this situation, we can select three of the channels to utilize.

5.9.1 RasterioSource

Any images that can be read by GDAL/Rasterio can be handled by the `RasterioSource`. This includes georeferenced imagery such as GeoTIFFs. If there are multiple image files that cover a single scene, you can pass the corresponding list of URIs, and read from the `RasterSource` as if it were a single stitched-together image.

The `RasterioSource` can also read non-georeferenced images such as `.tif`, `.png`, and `.jpg` files. This is useful for oblique drone imagery, biomedical imagery, and any other (potentially massive!) non-georeferenced images.

5.9.2 RasterizedSource

Semantic segmentation labels stored as polygons in a `VectorSource` can be rasterized and read using a `RasterizedSource`. This is a slightly unusual use of a `RasterSource` as we're using it to read labels, and not images to use as input to a model.

See also:

The *RasterioSourceConfig* and *RasterizedSourceConfig* API docs.

5.9.3 MultiRasterSource

A `RasterSource` that combines multiple sub-`RasterSources` by concatenting their outputs along the channel dimension (assumed to be the last dimension). This may be used, for example, to get RGB channels of a scene from one file and the elevation map of the same scene from another file, and then concatenate them together.

See also:

The api MultiRasterSourceConfig API docs.

5.10 RasterTransformer

A `RasterTransformer` is a mechanism for transforming raw raster data into a form that is more suitable for being fed into a model.

5.10.1 StatsTransformer

This transformer is used to convert non-uint8 values to uint8 using statistics computed by the *StatsAnalyzer*.

See also:

The *StatsTransformerConfig* API docs.

5.10.2 CastTransformer

This transformer is used to type-cast chips to a specific dtype. For example, to uint8.

See also:

The api CastTransformerConfig API docs.

5.10.3 NanTransformer

This transformer is used to remove NaN values from a float raster.

See also:

The api NanTransformerConfig API docs.

5.10.4 ReclassTransformer

This transformer is used to map labels in a label raster to different values.

See also:

The api ReclassTransformerConfig API docs.

5.11 VectorSource

A `VectorSource` supports reading vector data like polygons and lines from various places. It is used by `ObjectDetectionLabelSource` and `ChipClassificationLabelSource`, as well as the `RasterizedSource` (a type of `RasterSource`).

5.11.1 GeoJSONVectorSource

This vector source reads GeoJSON files.

See also:

The *GeoJSONVectorSourceConfig* API docs.

5.12 LabelSource

A `LabelSource` supports reading ground truth labels for a scene in the form of vectors or rasters. There are subclasses for different tasks and data formats. They can be queried for the labels that lie within a window and are used for creating training chips, as well as providing ground truth labels for evaluation against validation scenes.

See also:

The *ChipClassificationLabelSourceConfig*, *SemanticSegmentationLabelSourceConfig*, and *ObjectDetectionLabelSourceConfig* API docs.

5.12.1 LabelStore

A `LabelStore` supports reading and writing predicted labels for a scene. There are subclasses for different tasks and data formats. They are used for saving predictions and then loading them during evaluation.

In the `tiny_spacenet.py` example, there is no explicit `LabelStore` configured on the validation scene, because it can be inferred from the type of `RVPipelineConfig` it is part of. In the ISPRS Potsdam example, the following code is used to explicitly create a `LabelStore` that writes out the predictions in "RGB" format, where the color of each pixel represents the class, and predictions of class 0 (ie. car) are also written out as polygons.

```
label_store = SemanticSegmentationLabelStoreConfig(
    rgb=True, vector_output=[PolygonVectorOutputConfig(class_id=0)])

scene = SceneConfig(
    id=id,
    raster_source=raster_source,
    label_source=label_source,
    label_store=label_store)
```

See also:

The *ChipClassificationGeoJSONStoreConfig*, *SemanticSegmentationLabelStoreConfig*, and *ObjectDetectionGeoJSONStoreConfig* API docs.

5.13 Analyzers

Analyzers are used to gather dataset-level statistics and metrics for use in downstream processes. Typically, you won't need to explicitly configure any.

5.13.1 StatsAnalyzer

Currently the only analyzer available is the `StatsAnalyzer`, which determines the distribution of values over the imagery in order to normalize values to `uint8` values in a `StatsTransformer`.

5.14 Evaluators

For each computer vision task, there is an evaluator that computes metrics for a trained model. It does this by measuring the discrepancy between ground truth and predicted labels for a set of validation scenes. Typically, you won't need to explicitly configure any.

CHAPTER 6

Architecture and Customization

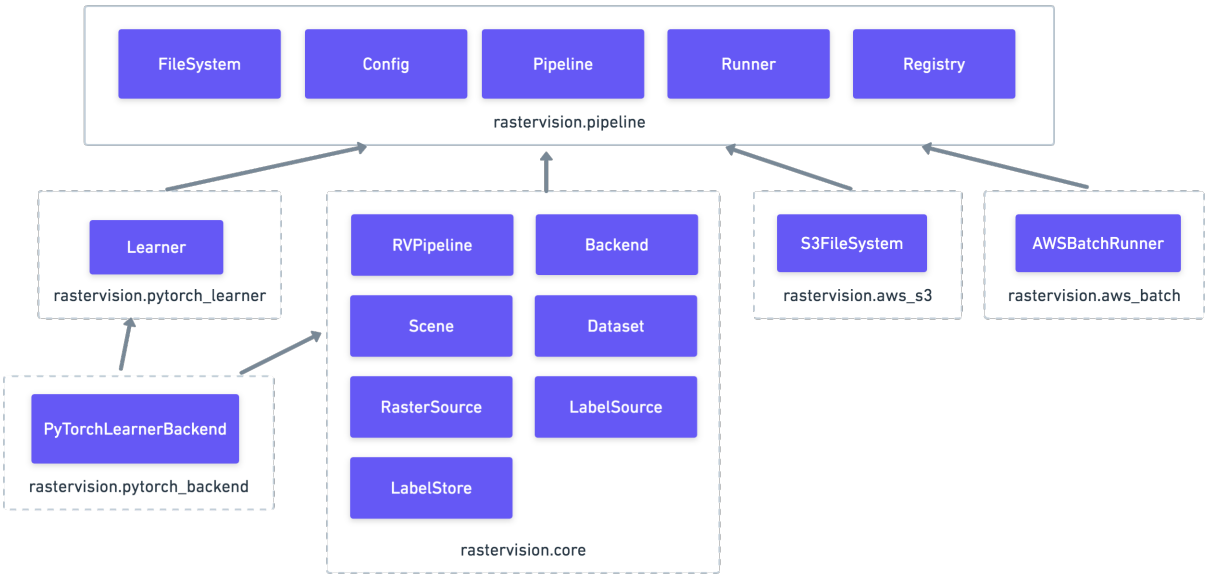
6.1 Codebase Overview

The Raster Vision codebase is designed with modularity and flexibility in mind. There is a main, required package, `rastervision.pipeline`, which contains functionality for defining and configuring computational pipelines, running them in different environments using parallelism and GPUs, reading and writing to different file systems, and adding and customizing pipelines via a plugin mechanism. In contrast, the "domain logic" of geospatial deep learning using PyTorch, and running on AWS is contained in a set of optional plugin packages. All plugin packages must be under the `rastervision` native namespace package.

Each of these packages is contained in a separate `setuptools`/`pip` package with its own dependencies, including dependencies on other Raster Vision packages. This means that it's possible to install and use subsets of the functionality in Raster Vision. A short summary of the packages is as follows:

- `rastervision.pipeline`: define and run pipelines
- `rastervision.aws_s3`: read and write files on S3
- `rastervision.aws_batch`: run pipelines on Batch
- `rastervision.core`: chip classification, object detection, and semantic segmentation pipelines that work on geospatial data along with abstractions for running with different *backends* and data formats
- `rastervision.pytorch_learner`: model building and training code using `torch` and `torchvision`, which can be used independently of `rastervision.core`.
- `rastervision.pytorch_backend`: adds backends for the pipelines in `rastervision.core` using `rastervision.pytorch_learner` to do the heavy lifting

The figure below shows the packages, the dependencies between them, and important base classes within each package.



6.2 Writing pipelines and plugins

In this section, we explain the most important aspects of the `rastervision.pipeline` package through a series of examples which incrementally build on one another. These examples show how to write custom pipelines and configuration schemas, how to customize an existing pipeline, and how to package the code as a plugin.

The full source code for Examples 1 and 2 is in rastervision.pipeline_example_plugin1 and Example 3 is in rastervision.pipeline_example_plugin2 and they can be run from inside the RV Docker image. However, **note that new plugins are typically created in a separate repo and Docker image**, and *Bootstrap new projects with a template* shows how to do this.

6.2.1 Example 1: a simple pipeline

A `Pipeline` in RV is a class which represents a sequence of commands with a shared configuration in the form of a `PipelineConfig`. Here is a toy example of these two classes that saves a set of messages to disk, and then prints them all out.

Listing 1: rastervision.pipeline_example_plugin1.sample_pipeline

```
from typing import List, Optional
from os.path import join

from rastervision.pipeline.pipeline import Pipeline
from rastervision.pipeline.file_system import str_to_file, file_to_str
from rastervision.pipeline.pipeline_config import PipelineConfig
from rastervision.pipeline.config import register_config
from rastervision.pipeline.utils import split_into_groups


# Each Config needs to be registered with a type hint which is used for
# serializing and deserializing to JSON.
@register_config('pipeline_example_plugin1.sample_pipeline')
```

(continues on next page)

(continued from previous page)

```
class SamplePipelineConfig(PipelineConfig):
    # Config classes are configuration schemas. Each field is an attributes
    # with a type and optional default value.
    names: List[str] = ['alice', 'bob']
    message_uris: Optional[List[str]] = None

    def build(self, tmp_dir):
        # The build method is used to instantiate the corresponding object
        # using this configuration.
        return SamplePipeline(self, tmp_dir)

    def update(self):
        # The update method is used to set default values as a function of
        # other values.
        if self.message_uris is None:
            self.message_uris = [
                join(self.root_uri, '{}.txt'.format(name))
                for name in self.names
            ]


class SamplePipeline(Pipeline):
    # The order in which commands run. Each command correspond to a method.
    commands: List[str] = ['save_messages', 'print_messages']

    # Split commands can be split up and run in parallel.
    split_commands = ['save_messages']

    # GPU commands are run using GPUs if available. There are no commands worth␣
↪running
    # on a GPU in this pipeline.
    gpu_commands = []

    def save_messages(self, split_ind=0, num_splits=1):
        # Save a file for each name with a message.

        # The num_splits is the number of parallel jobs to use and
        # split_ind tracks the index of the parallel job. In this case
        # we are splitting on the names/message_uris.
        split_groups = split_into_groups(
            list(zip(self.config.names, self.config.message_uris)), num_splits)
        split_group = split_groups[split_ind]

        for name, message_uri in split_group:
            message = 'hello {}!'.format(name)
            # str_to_file and most functions in the file_system package can
            # read and write transparently to different file systems based on
            # the URI pattern.
            str_to_file(message, message_uri)
            print('Saved message to {}'.format(message_uri))

    def print_messages(self):
        # Read all the message files and print them.
        for message_uri in self.config.message_uris:
            message = file_to_str(message_uri)
            print(message)
```

In order to run this, we need a separate Python file with a `get_config()` function which provides an instantiation of the `SamplePipelineConfig`.

Listing 2: rastervision.pipeline_example_plugin1.config1

```
from rastervision.pipeline_example_plugin1.sample_pipeline import (
    SamplePipelineConfig)


def get_config(runner, root_uri):
    # The get_config function returns an instantiated PipelineConfig and
    # plays a similar role as a typical "config file" used in other systems.
    # It's different in that it can have loops, conditionals, local variables,
    # etc. The runner argument is the name of the runner used to run the
    # pipeline (eg. local or batch). Any other arguments are passed from the
    # CLI using the -a option.
    names = ['alice', 'bob', 'susan']

    # Note that root_uri is a field that is inherited from PipelineConfig,
    # the parent class of SamplePipelineConfig, and specifies the root URI
    # where any output files are saved.
    return SamplePipelineConfig(root_uri=root_uri, names=names)
```

Finally, in order to package this code as a plugin, and make it usable within the Raster Vision framework, it needs to be in a package directly under the `rastervision` namespace package, and have a top-level `__init__.py` file with a certain structure.

Listing 3: rastervision.pipeline_example_plugin1.__init__

```
# flake8: noqa

# Must import pipeline package first.
import rastervision.pipeline

# Then import any modules that add Configs so that the register_config decorators
# get called.
import rastervision.pipeline_example_plugin1.sample_pipeline
import rastervision.pipeline_example_plugin1.sample_pipeline2


def register_plugin(registry):
    # Can be used to manually update the registry. Useful
    # for adding new FileSystems and Runners.
    pass
```

We can invoke the Raster Vision CLI to run the pipeline using:

```
> rastervision run inprocess rastervision.pipeline_example_plugin1.config1 -a root_
↪uri /opt/data/pipeline-example/1/ -s 2

Running save_messages command split 1/2...
Saved message to /opt/data/pipeline-example/1/alice.txt
Saved message to /opt/data/pipeline-example/1/bob.txt
Running save_messages command split 2/2...
Saved message to /opt/data/pipeline-example/1/susan.txt
Running print_messages command...
hello alice!
hello bob!
```

(continues on next page)

(continued from previous page)

```
hello susan!
```

This uses the `inprocess` runner, which executes all the commands in a single process locally (which is good for debugging), and uses the `LocalFileSystem` to read and write files. The `-s 2` option says to use two splits for splittable commands, and the `-a root_uri /opt/data/sample-pipeline` option says to pass the `root_uri` argument to the `get_config` function.

6.2.2 Example 2: hierarchical config

This example makes some small changes to the previous example, and shows how configurations can be built up hierarchically. However, the main purpose here is to lay the foundation for *Example 3: customizing an existing pipeline* which shows how to customize the configuration schema and behavior of this pipeline using a plugin. The changes to the previous example are highlighted with comments, but the overall effect is to delegate making messages to a `MessageMaker` class with its own `MessageMakerConfig` including a `greeting` field.

Listing 4: rastervision.pipeline_example_plugin1.sample_pipeline2

```
from typing import List, Optional
from os.path import join

from rastervision.pipeline.pipeline import Pipeline
from rastervision.pipeline.file_system import str_to_file, file_to_str
from rastervision.pipeline.pipeline_config import PipelineConfig
from rastervision.pipeline.config import register_config, Config
from rastervision.pipeline.utils import split_into_groups


@register_config('pipeline_example_plugin1.message_maker')
class MessageMakerConfig(Config):
    greeting: str = 'hello'

    def build(self):
        return MessageMaker(self)


class MessageMaker():
    def __init__(self, config):
        self.config = config

    def make_message(self, name):
        # Use the greeting field to make the message.
        return '{} {}!'.format(self.config.greeting, name)


@register_config('pipeline_example_plugin1.sample_pipeline2')
class SamplePipeline2Config(PipelineConfig):
    names: List[str] = ['alice', 'bob']
    message_uris: Optional[List[str]] = None
    # Fields can have other Configs as types.
    message_maker: MessageMakerConfig = MessageMakerConfig()

    def build(self, tmp_dir):
        return SamplePipeline2(self, tmp_dir)

    def update(self):
```

(continues on next page)

(continued from previous page)

```
        if self.message_uris is None:
            self.message_uris = [
                join(self.root_uri, '{}.txt'.format(name))
                for name in self.names
            ]


class SamplePipeline2(Pipeline):
    commands: List[str] = ['save_messages', 'print_messages']
    split_commands = ['save_messages']
    gpu_commands = []

    def save_messages(self, split_ind=0, num_splits=1):
        message_maker = self.config.message_maker.build()

        split_groups = split_into_groups(
            list(zip(self.config.names, self.config.message_uris)), num_splits)
        split_group = split_groups[split_ind]

        for name, message_uri in split_group:
            # Unlike before, we use the message_maker to make the message.
            message = message_maker.make_message(name)
            str_to_file(message, message_uri)
            print('Saved message to {}'.format(message_uri))

    def print_messages(self):
        for message_uri in self.config.message_uris:
            message = file_to_str(message_uri)
            print(message)
```

We can configure the pipeline using:

Listing 5: rastervision.pipeline_example_plugin1.config2

```
from rastervision.pipeline_example_plugin1.sample_pipeline2 import (
    SamplePipeline2Config, MessageMakerConfig)


def get_config(runner, root_uri):
    names = ['alice', 'bob', 'susan']
    # Same as before except we can set the greeting to be
    # 'hola' instead of 'hello'.
    message_maker = MessageMakerConfig(greeting='hola')
    return SamplePipeline2Config(
        root_uri=root_uri, names=names, message_maker=message_maker)
```

The pipeline can then be run with the above configuration using:

```
> rastervision run inprocess rastervision.pipeline_example_plugin1.config2 -a root_
↪uri /opt/data/pipeline-example/2/ -s 2

Running save_messages command split 1/2...
Saved message to /opt/data/pipeline-example/2/alice.txt
Saved message to /opt/data/pipeline-example/2/bob.txt
Running save_messages command split 2/2...
Saved message to /opt/data/pipeline-example/2/susan.txt
Running print_messages command...
```

(continues on next page)

(continued from previous page)

```
hola alice!
hola bob!
hola susan!
```

6.2.3 Example 3: customizing an existing pipeline

This example shows how to customize the behavior of an existing pipeline, namely the `SamplePipeline2` developed in *Example 2: hierarchical config*. That pipeline delegates printing messages to a `MessageMaker` class which is configured by `MessageMakerConfig`. Our goal here is to make it possible to control the number of exclamation points at the end of the message.

By writing a plugin (ie. a plugin to the existing plugin that was developed in the previous two examples), we can add new behavior without modifying any of the original source code from *Example 2: hierarchical config*. This mimics the situation plugin writers will be in when they want to modify the behavior of one of the *geospatial deep learning pipelines* without modifying the source code in the main Raster Vision repo.

The code to implement the new configuration and behavior, and a sample configuration are below. (We omit the `__init__.py` file since it is similar to the one in the previous plugin.) Note that the new `DeluxeMessageMakerConfig` uses inheritance to extend the configuration schema.

Listing 6: rastervision.pipeline_example_plugin2.deluxe_message_maker

```
from rastervision.pipeline.config import register_config
from rastervision.pipeline_example_plugin1.sample_pipeline2 import (
    MessageMakerConfig, MessageMaker)


# You always need to use the register_config decorator.
@register_config('pipeline_example_plugin2.deluxe_message_maker')
class DeluxeMessageMakerConfig(MessageMakerConfig):
    # Note that this inherits the greeting field from MessageMakerConfig.
    level: int = 1

    def build(self):
        return DeluxeMessageMaker(self)


class DeluxeMessageMaker(MessageMaker):
    def make_message(self, name):
        # Uses the level field to determine the number of exclamation marks.
        exclamation_marks = '!' * self.config.level
        return '{} {}{}'.format(self.config.greeting, name, exclamation_marks)
```

Listing 7: rastervision.pipeline_example_plugin2.config3

```
from rastervision.pipeline_example_plugin1.sample_pipeline2 import (
    SamplePipeline2Config)
from rastervision.pipeline_example_plugin2.deluxe_message_maker import (
    DeluxeMessageMakerConfig)


def get_config(runner, root_uri):
    names = ['alice', 'bob', 'susan']
    # Note that we use the DeluxeMessageMakerConfig and set the level to 3.
    message_maker = DeluxeMessageMakerConfig(greeting='hola', level=3)
```

(continues on next page)

(continued from previous page)

```
    return SamplePipeline2Config(
        root_uri=root_uri, names=names, message_maker=message_maker)
```

We can run the pipeline as follows:

```
> rastervision run inprocess rastervision.pipeline_example_plugin2.config3 -a root_
↪uri /opt/data/pipeline-example/3/ -s 2

Running save_messages command split 1/2...
Saved message to /opt/data/pipeline-example/3/alice.txt
Saved message to /opt/data/pipeline-example/3/bob.txt
Running save_messages command split 2/2...
Saved message to /opt/data/pipeline-example/3/susan.txt
Running print_messages command...
hola alice!!!
hola bob!!!
hola susan!!!
```

The output in `/opt/data/sample-pipeline` contains a `pipeline-config.json` file which is the serialized version of the `SamplePipeline2Config` created in `config3.py`. The serialized configuration is used to transmit the configuration when running a pipeline remotely. It also is a programming language-independent record of the fully-instantiated configuration that was generated by the `run` command in conjunction with any command line arguments. Below is the partial contents of this file. The interesting thing to note here is the `type_hint` field that appears twice. This is what allows the JSON to be deserialized back into the Python classes that were originally used.(Recall that the `register_config` decorator is what tells the `Registry` the type hint for each `Config` class.)

```
{
    "root_uri": "/opt/data/sample-pipeline",
    "type_hint": "sample_pipeline2",
    "names": [
        "alice",
        "bob",
        "susan"
    ],
    "message_uris": [
        "/opt/data/sample-pipeline/alice.txt",
        "/opt/data/sample-pipeline/bob.txt",
        "/opt/data/sample-pipeline/susan.txt"
    ],
    "message_maker": {
        "greeting": "hola",
        "type_hint": "deluxe_message_maker",
        "level": 3
    }
}
```

We now have a plugin that customizes an existing pipeline! Being a toy example, this may all seem like overkill. Hopefully, the real power of the `pipeline` package becomes more apparent when considering the standard set of plugins distributed with Raster Vision, and how this functionality can be customized with user-created plugins.

6.3 Customizing Raster Vision

When approaching a new problem or dataset with Raster Vision, you may get lucky and be able to apply RV "off-the-shelf". In other cases, Raster Vision can be used after writing scripts to convert data into the appropriate format.

However, sometimes you will need to modify the functionality of RV to suit your problem. In this case, you could modify the RV source code (ie. any of the code in the *packages* in the main RV repo). In some cases, this may be necessary, as the right extension points don't exist. In other cases, the functionality may be very widely-applicable, and you would like to *contribute* it to the main repo. Most of the time, however, the functionality will be problem-specific, or is in an embryonic stage of development, and should be implemented in a plugin that resides outside the main repo.

General information about plugins can be found in *Bootstrap new projects with a template* and *Writing pipelines and plugins*. The following are some brief pointers on how to write plugins for different scenarios. In the future, we would like to enhance this section.

- To add commands to an existing `Pipeline`: write a plugin with subclasses of the `Pipeline` and its corresponding `PipelineConfig` class. The new `Pipeline` should add a method for the new command, and modify the list of `commands`. Any new configuration should be added to the subclass of the `PipelineConfig`. Example: running some data pre- or post-processing code in a pipeline.
- To modify commands of an existing `Pipeline`: same as above except you will override command methods. If a new configuration field is required, you can subclass the `Config` class that field resides within. Example: custom chipping functionality for semantic segmentation. You will need to create subclasses of `SemanticSegmentationChipOptions`, `SemanticSegmentation`, and `SemanticSegmentationConfig`.
- To create a substantially new `Pipeline`: write a new plugin that adds a new `Pipeline`. See the `rastervision.core` plugin, in particular, the contents of the `rastervision.core.rv_pipeline` package. If you want to add a new geospatial deep learning pipeline (eg. for chip regression), you may want to override the `RVPipeline` class. In other cases that deviate more from `RVPipeline`, you may want to write a new `Pipeline` class with arbitrary commands and logic, but that uses the core model building and training functionality in the `rastervision.pytorch_learner` plugin.
- To add the ability to use new file systems or run in new cloud environments: write a plugin that adds a new `FileSystem` or `Runner`. See the `rastervision.aws_s3` and `rastervision.aws_batch` plugins for examples.
- To use an existing `RVPipeline` with a new `Backend`: write a plugin that adds a subclass of `Backend` and `BackendConfig`. See the `rastervision.pytorch_backend` plugin for an example.
- To override model building or training routines in an existing `PyTorchLearnerBackend`: write a plugin that adds a subclass of `Learner` (and `LearnerConfig`) that overrides `build_model` and `train_step`, and a subclass of `PyTorchLearnerBackend` (and `PyTorchLearnerBackendConfig`) that overrides the backend so it uses the `Learner` subclass.

CHAPTER 7

Examples

This page contains examples of using Raster Vision on open datasets. Unless otherwise stated, all commands should be run inside the Raster Vision Docker container. See *Docker Images* for info on how to do this.

7.1 How to Run an Example

There is a common structure across all of the examples which represents a best practice for defining experiments. Running an example involves the following steps.

- Acquire raw dataset.
- (Optional) Get processed dataset which is derived from the raw dataset, either using a Jupyter notebook, or by downloading the processed dataset.
- (Optional) Do an abbreviated test run of the experiment on a small subset of data locally.
- Run full experiment on GPU.
- Inspect output
- (Optional) Make predictions on new imagery

Each of the examples has several arguments that can be set on the command line:

- The input data for each experiment is divided into two directories: the raw data which is publicly distributed, and the processed data which is derived from the raw data. These two directories are set using the `raw_uri` and `processed_uri` arguments.
- The output generated by the experiment is stored in the directory set by the `root_uri` argument.
- The `raw_uri`, `processed_uri`, and `root_uri` can each be local or remote (on S3), and don't need to agree on whether they are local or remote.
- Experiments have a `test` argument which runs an abbreviated experiment for testing/debugging purposes.

In the next section, we describe in detail how to run one of the examples, SpaceNet Rio Chip Classification. For other examples, we only note example-specific details.

7.2 Chip Classification: SpaceNet Rio Buildings

This example performs chip classification to detect buildings on the Rio AOI of the SpaceNet dataset.

7.2.1 Step 1: Acquire Raw Dataset

The dataset is stored on AWS S3 at `s3://spacenet-dataset`. You will need an AWS account to access this dataset, but it will not be charged for accessing it. (To forward you AWS credentials into the container, use `docker/run --aws`).

Optional: to run this example with the data stored locally, first copy the data using something like the following inside the container.

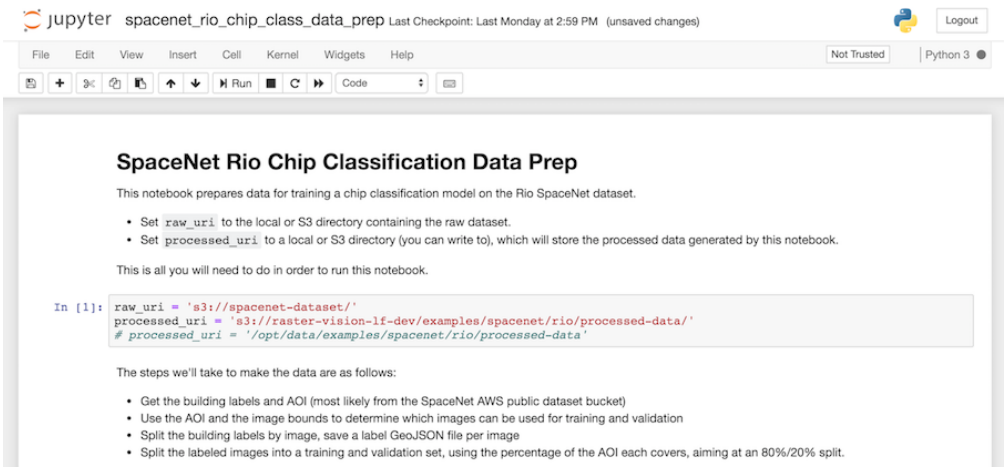
```
aws s3 sync s3://spacenet-dataset/AOIs/AOI_1_Rio/ /opt/data/spacenet-dataset/AOIs/AOI_
↪1_Rio/
```

7.2.2 Step 2: Run the Jupyter Notebook

You'll need to do some data preprocessing, which we can do in the Jupyter notebook supplied.

```
docker/run --jupyter [--aws]
```

The `--aws` option is only needed if pulling data from S3. In Jupyter inside the browser, navigate to the rastervision/examples/chip_classification/spacenet_rio_data_prep.ipynb notebook. Set the URIs in the first cell and then run the rest of the notebook. Set the `processed_uri` to a local or S3 URI depending on where you want to run the experiment.



7.2.3 Step 3: Do a test run locally

The experiment we want to run is in spacenet_rio.py. To run this, first get to the Docker console using:

```
docker/run [--aws] [--gpu] [--tensorboard]
```

The `--aws` option is only needed if running experiments on AWS or using data stored on S3. The `--gpu` option should only be used if running on a local GPU. The `--tensorboard` option should be used if running locally and you would like to view Tensorboard. The test run can be executed using something like:

```
export RAW_URI="s3://spacenet-dataset/"
export PROCESSED_URI="/opt/data/examples/spacenet/rio/processed-data"
export ROOT_URI="/opt/data/examples/spacenet/rio/local-output"

rastervision run local rastervision.examples.chip_classification.spacenet_rio \
    -a raw_uri $RAW_URI -a processed_uri $PROCESSED_URI -a root_uri $ROOT_URI \
    -a test True --splits 2
```

The sample above assumes that the raw data is on S3, and the processed data and output are stored locally. The `raw_uri` directory is assumed to contain an `AOIs/AOI_1_Rio` subdirectory. This runs two parallel jobs for the `chip` and `predict` commands via `--splits 2`. See `rastervision --help` and `rastervision run --help` for more usage information.

Note that when running with `-a test True`, some crops of the test scenes are created and stored in `processed_uri/crops/`. All of the examples that use big image files use this trick to make the experiment run faster in test mode.

After running this, the main thing to check is that it didn't crash, and that the visualization of training and validation chips look correct. These "debug chips" for each of the data splits can be found in `$ROOT_URI/train/dataloaders/`.

7.2.4 Step 4: Run full experiment

To run the full experiment on GPUs using AWS Batch, use something like the following. Note that all the URIs are on S3 since remote instances will not have access to your local file system.

```
export RAW_URI="s3://spacenet-dataset/"
export PROCESSED_URI="s3://mybucket/examples/spacenet/rio/processed-data"
export ROOT_URI="s3://mybucket/examples/spacenet/rio/remote-output"

rastervision run batch rastervision.examples.chip_classification.spacenet_rio \
    -a raw_uri $RAW_URI -a processed_uri $PROCESSED_URI -a root_uri $ROOT_URI \
    -a test False --splits 8
```

For instructions on setting up AWS Batch resources and configuring Raster Vision to use them, see *Setting up AWS Batch*. To monitor the training process using Tensorboard, visit `<public dns>:6006` for the EC2 instance running the training job.

If you would like to run on a local GPU, replace `batch` with `local`, and use local URIs. To monitor the training process using Tensorboard, visit `localhost:6006`, assuming you used `docker/run --tensorboard`.

7.2.5 Step 5: Inspect results

After everything completes, which should take about 1.5 hours if you're running on AWS using a `p3.2xlarge` instance for training and 8 splits, you should be able to find the predictions over the validation scenes in `$root_uri/predict/`. The imagery and predictions are best viewed in QGIS, an example of which can be seen below. Cells that are predicted to contain buildings are red, and background are green.

The evaluation metrics can be found in `$root_uri/eval/eval.json`. This is an example of the scores from a run, which show an F1 score of 0.97 for detecting chips with buildings.


```
[
    {
        "precision": 0.9802512682554008,
        "recall": 0.9865974924340684,
        "f1": 0.9833968183611386,
        "count_error": 0.0,
        "gt_count": 2313.0,
        "class_id": 0,
        "class_name": "no_building"
    },
    {
        "precision": 0.9789227645464389,
        "recall": 0.9685147159479809,
        "f1": 0.9736038795756798,
        "count_error": 0.0,
        "gt_count": 1461.0,
        "class_id": 1,
        "class_name": "building"
    },
    {
        "precision": 0.9797369746892128,
        "recall": 0.9795972443031267,
        "f1": 0.9796057522335405,
        "count_error": 0.0,
        "gt_count": 3774.0,
        "class_id": null,
        "class_name": "average"
```


(continues on next page)

(continued from previous page)

```
    }
]
```

More evaluation details can be found here.

7.2.6 Step 6: Predict on new imagery

After running an experiment, a **model bundle** is saved into `$root_uri/bundle/`. This can be used to make predictions on new images. See the *Model Zoo* section for more details.

7.3 Semantic Segmentation: SpaceNet Vegas

This experiment contains an example of doing semantic segmentation using the SpaceNet Vegas dataset which has labels in vector form. It allows for training a model to predict buildings or roads. Note that for buildings, polygon output in the form of GeoJSON files will be saved to the `predict` directory alongside the GeoTIFF files. In addition, a vector evaluation file using SpaceNet metrics will be saved to the `eval` directory.

Arguments:

- `raw_uri` should be set to the root of the SpaceNet data repository, which is at `s3://spacenet-dataset`, or a local copy of it. A copy only needs to contain the `AOIs/AOI_2_Vegas` subdirectory.
- `target` can be `buildings` or `roads`
- `processed_uri` should not be set because there is no processed data in this example.

7.3.1 Buildings

After training a model, the building F1 score is 0.91. More evaluation details can be found here.

7.3.2 Roads

After training a model, the road F1 score was 0.83. More evaluation details can be found here.

7.4 Semantic Segmentation: ISPRS Potsdam

This experiment performs semantic segmentation on the ISPRS Potsdam dataset. The dataset consists of 5cm aerial imagery over Potsdam, Germany, segmented into six classes including building, tree, low vegetation, impervious, car, and clutter. For more info see our blog post.

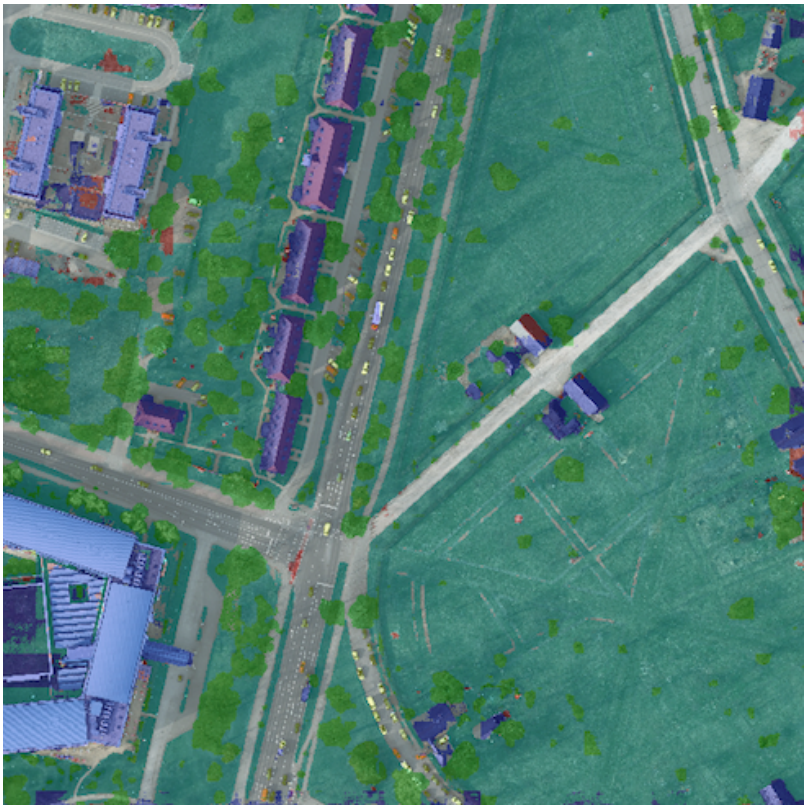
Data:

- The dataset can only be downloaded after filling in this request form. After your request is granted, follow the link to 'POTSDAM 2D LABELING' and download and unzip `4_Ortho_RGBIR.zip`, and `5_Labels_for_participants.zip` into a directory, and then upload to S3 if desired.

Arguments:

- `raw_uri` should contain `4_Ortho_RGBIR` and `5_Labels_for_participants` subdirectories.
- `processed_uri` should be set to a directory which will be used to store test crops.

After training a model, the average F1 score was 0.89. More evaluation details can be found here.



7.5 Object Detection: COWC Potsdam Cars

This experiment performs object detection on cars with the Cars Overhead With Context dataset over Potsdam, Germany.

Data:

- The imagery can only be downloaded after filling in this request form. After your request is granted, follow the link to 'POTSDAM 2D LABELING' and download and unzip `4_Ortho_RGBIR.zip` into a directory, and then upload to S3 if desired. (This step uses the same imagery as *Semantic Segmentation: ISPRS Potsdam*.)

- Download the processed labels and unzip. These files were generated from the COWC car detection dataset using some scripts. TODO: Get these scripts into runnable shape.

Arguments:

- `raw_uri` should point to the imagery directory created above, and should contain the `4_Ortho_RGBIR` subdirectory.
- `processed_uri` should point to the labels directory created above. It should contain the `labels/all` subdirectory.

After training a model, the car F1 score was 0.95. More evaluation details can be found here.

7.6 Object Detection: xView Vehicles

This experiment performs object detection to find vehicles using the DIUx xView Detection Challenge dataset.

Data:

- Sign up for an account for the DIUx xView Detection Challenge. Navigate to the downloads page and download the zipped training images and labels. Unzip both of these files and place their contents in a directory, and upload to S3 if desired.
- Run the xview-data-prep.ipynb Jupyter notebook, pointing the `raw_uri` to the directory created above.

Arguments:

- The `raw_uri` should point to the directory created above, and contain a labels GeoJSON file named `xView_train.geojson`, and a directory named `train_images`.
- The `processed_uri` should point to the processed data generated by the notebook.

After training a model, the vehicle F1 score was 0.61. More evaluation details can be found here.

7.7 Model Zoo

Using the Model Zoo, you can download model bundles which contain pre-trained models and meta-data, and then run them on sample test images that the model wasn't trained on.

```
rastervision predict <model bundle> <infile> <outfile>
```

Note that the input file is assumed to have the same channel order and statistics as the images the model was trained on. See `rastervision predict --help` to see options for manually overriding these. It shouldn't take more than a minute on a CPU to make predictions for each sample. For some of the examples, there are also model files that can be used for fine-tuning on another dataset.

Disclaimer: These models are provided for testing and demonstration purposes and aren't particularly accurate. As is usually the case for deep learning models, the accuracy drops greatly when used on input that is outside the training distribution. In other words, a model trained on one city probably won't work well on another city (unless they are very similar) or at a different imagery resolution.

When unzipped, the model bundle contains a `model.pth` file which can be used for fine-tuning.

Table 1: Model Zoo

Dataset	Task	Model Type	Model Bundle	Sample Image
SpaceNet Rio Buildings	Chip Classification	Resnet 50	link	link
SpaceNet Vegas Buildings	Semantic Segmentation	DeeplabV3 / Resnet50	link	link
SpaceNet Vegas Roads	Semantic Segmentation	DeeplabV3 / Resnet50	link	link
ISPRS Potsdam	Semantic Segmentation	Panoptic FPN / Resnet50	link	link
COWC Potsdam (Cars)	Object Detection	Faster-RCNN / Resnet18	link	link
xView (Vehicles)	Object Detection	Faster-RCNN / Resnet50	link	link

CHAPTER 8

Bootstrap new projects with a template

When using Raster Vision on a new project, the best practice is to create a new repo with its own Docker image based on the Raster Vision image. This involves a fair amount of boilerplate code which has a few things that vary between projects. To facilitate bootstrapping new projects, there is a cookiecutter template. Assuming that you cloned the Raster Vision repo and ran `pip install cookiecutter==1.7.0`, you can instantiate the template as follows (after adjusting paths appropriately for your particular setup).

```
[lfishgold@monoshone ~/projects]
$ cookiecutter raster-vision/cookiecutter_template/
caps_project_name [MY_PROJECT]:
project_name [my_project]:
docker_image [my_project]:
parent_docker_image [quay.io/azavea/raster-vision:pytorch-0.13]:
version [0.13]:
description [A Raster Vision plugin]:
url [https://github.com/azavea/raster-vision]:
author [Azavea]:
author_email [info@azavea.com]:

[lfishgold@monoshone ~/projects]
$ tree my_project/
my_project/
├── Dockerfile
├── README.md
├── docker
│   ├── build
│   ├── ecr_publish
│   └── run
└── rastervision_my_project
    ├── rastervision
    │   └── my_project
    │       ├── __init__.py
    │       ├── configs
    │       │   ├── __init__.py
    │       │   └── test.py
```

(continues on next page)

(continued from previous page)

```
    │       ├── test_pipeline.py
    │       └── test_pipeline_config.py
    ├── requirements.txt
    └── setup.py

5 directories, 12 files
```

The output is a repo structure with the skeleton of a Raster Vision plugin that can be pip installed, and everything needed to build, run, and publish a Docker image with the plugin. The resulting `README.md` file contains setup and usage information for running locally and on Batch, which makes use of the *CloudFormation setup* for creating new user/project-specific job defs.

CHAPTER 9

Setup AWS Batch using CloudFormation

This describes the deployment code that sets up the necessary AWS resources to utilize the AWS Batch runner. Using Batch is advantageous because it starts and stops instances automatically and runs jobs sequentially or in parallel according to the dependencies between them. In addition, this deployment sets up distinct CPU and GPU resources and utilizes spot instances, which is more cost-effective than always using a GPU on-demand instance. Deployment is driven via the AWS console using a CloudFormation template.

This AWS Batch setup is an "advanced" option that assumes some familiarity with Docker, AWS IAM, named profiles, availability zones, EC2, ECR, CloudFormation, and Batch.

9.1 AWS Account Setup

In order to setup Batch using this repo, you will need to setup your AWS account so that:

- you have either root access to your AWS account, or an IAM user with admin permissions. It is probably possible with less permissions, but we haven't figured out how to do this yet after some experimentation.
- you have the ability to launch P2 or P3 instances which have GPUs.
- you have requested permission from AWS to use availability zones outside the USA if you would like to use them. (New AWS accounts can't launch EC2 instances in other AZs by default.) If you are in doubt, just use `us-east-1`.

9.2 AWS Credentials

Using the AWS CLI, create an AWS profile for the target AWS environment. An example, naming the profile `raster-vision`:

```
$ aws --profile raster-vision configure
AWS Access Key ID [****************F2DQ]:
AWS Secret Access Key [****************TLJ/]:
```

(continues on next page)

(continued from previous page)

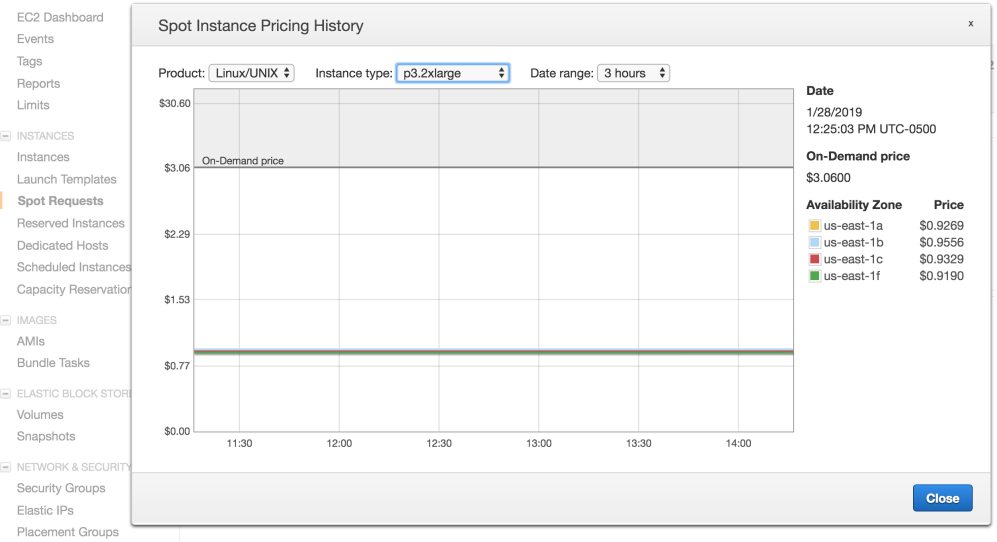
```
Default region name [us-east-1]: us-east-1
Default output format [None]:
```

You will be prompted to enter your AWS credentials, along with a default region. The Access Key ID and Secret Access Key can be retrieved from the IAM console. These credentials will be used to authenticate calls to the AWS API when using Packer and the AWS CLI.

9.3 Deploying Batch resources

To deploy AWS Batch resources using AWS CloudFormation, start by logging into your AWS console. Then, follow the steps below:

- Navigate to `CloudFormation > Create Stack`
- In the `Choose a template field`, select `Upload a template to Amazon S3` and upload the template in cloudformation/template.yml. **Warning:** Some versions of Chrome fail at this step without an explanation. As a workaround, try a different version of Chrome, or Firefox. See this thread for more details.
- `Prefix`: If you are setting up multiple RV stacks within an AWS account, you need to set a prefix for namespacing resources. Otherwise, there will be name collisions with any resources that were created as part of another stack.
- **Specify the following required parameters:**
  - `Stack Name`: The name of your CloudFormation stack
  - `VPC`: The ID of the Virtual Private Cloud in which to deploy your resource. Your account should have at least one by default.
  - `Subnets`: The ID of any subnets that you want to deploy your resources into. Your account should have at least two by default; make sure that the subnets you select are in the VPC that you chose by using the AWS VPC console, or else CloudFormation will throw an error. (Subnets are tied to availability zones, and so affect spot prices.) In addition, you need to choose subnets that are available for the instance type you have chosen. To find which subnets are available, go to Spot Pricing History in the EC2 console and select the instance type. Then look up the availability zones that are present in the VPC console to find the corresponding subnets. Your spot requests will be more likely to be successful and your savings will be greater if you have subnets in more availability zones.



  - `SSH Key Name`: The name of the SSH key pair you want to be able to use to shell into your Batch instances. If you've created an EC2 instance before, you should already have one you can use; otherwise, you can create one in the EC2 console. *Note: If you decide to create a new one, you will need to log out and then back in to the console before creating a Cloudformation stack using this key.*
  - `Instance Types`: Provide the instance types you would like to use. (For GPUs, `p3.2xlarge` is approximately 4 times the speed for 4 times the price.)

- **Adjust any preset parameters that you want to change (the defaults should be fine for most users) and click `Next`.**

  - Advanced users: If you plan on modifying Raster Vision and would like to publish a custom image to run on Batch, you will need to specify an ECR repo name and a tag name. Note that the repo names cannot be the same as the Stack name (the first field in the UI) and cannot be the same as any existing ECR repo names. If you are in a team environment where you are sharing the AWS account, the repo names should contain an identifier such as your username.

- Accept all default options on the `Options` page and click `Next`
- Accept "I acknowledge that AWS CloudFormation might create IAM resources with custom names" on the `Review` page and click `Create`
- Watch your resources get deployed!

9.4 Publish local Raster Vision images to ECR

If you setup ECR repositories during the CloudFormation setup (the "advanced user" option), then you will need to follow this step, which publishes local Raster Vision images to those ECR repositories. Every time you make a change to your local Raster Vision images and want to use those on Batch, you will need to run these steps:

- Run `./docker/build` in the Raster Vision repo to build a local copy of the Docker image.
- Run `./docker/ecr_publish` in the Raster Vision repo to publish the Docker images to ECR. Note that this requires setting the `RV_ECR_IMAGE` environment variable to be set to `<ecr_repo_name>:<tag_name>`.

9.5 Update Raster Vision configuration

Finally, make sure to update your *Setting up AWS Batch* with the Batch resources that were created.

9.6 Deploy new job definitions

When a user starts working on a new RV-based project (or a new user starts working on an existing RV-based project), they will often want to publish a custom Docker image to ECR and use it when running on Batch. To facilitate this, there is a separate cloudformation/job_def_template.yml. The idea is that for each user/project pair which is identified by a `Namespace` string, a CPU and GPU job definition is created which point to a specified ECR repo using that `Namespace` as the tag. After creating these new resources, the image should be published to `<repo>:<namespace>` on ECR, and the new job definitions should be placed in a project-specific RV profile file.

CHAPTER 10

Running Pipelines

Running pipelines in Raster Vision is done using the `rastervision` *run* command. This generates a pipeline configuration, serializes it, and then uses a runner to actually execute the commands, locally or remotely.

See also:

pipeline package explains more of the details of how `Pipelines` are implemented.

10.1 Running locally

10.1.1 local

A `rastervision run local ...` command will use the `LocalRunner`, which builds a `Makefile` based on the pipeline and executes it on the host machine. This will run multiple pipelines in parallel, as well as splittable commands in parallel, by spawning new processes for each command, where each process runs `rastervision run_command ...`.

10.1.2 inprocess

For debugging purposes, using `rastervision run inprocess` will run everything sequentially within a single process.

10.2 Running remotely

10.2.1 batch

Running `rastervision run batch ...` will submit a DAG (directed acyclic graph) of jobs to be run on AWS Batch, which will increase the instance count to meet the workload with low-cost spot instances, and terminate the instances when the queue of commands is finished. It can also run some commands on CPU instances (like `chip`),

and others on GPU (like `train`), and will run multiple experiments in parallel, as well as splittable commands in parallel.

The `AWSBatchRunner` executes each command by submitting a job to Batch, which executes the `rastervision run_command` inside the Docker image configured in the Batch job definition. Commands that are dependent on an upstream command are submitted as a job after the upstream command's job, with the `jobId` of the upstream command job as the parent `jobId`. This way Batch knows to wait to execute each command until all upstream commands are finished executing, and will fail the command if any upstream commands fail.

If you are running on AWS Batch or any other remote runner, you will not be able to use your local file system to store any of the data associated with an experiment.

Note: To run on AWS Batch, you'll need the proper setup. See *Setting up AWS Batch* for instructions.

10.3 Running Commands in Parallel

Raster Vision can run certain commands in parallel, such as the *CHIP* and *PREDICT* commands. These commands are designated as `split_commands` in the corresponding `Pipeline` class. To run split commands in parallel, use the `--split` option to the *run* CLI command.

Splittable commands can be run in parallel, with each instance doing its share of the workload. For instance, using `--splits 5` on a `CHIP` command over 50 training scenes and 25 validation scenes will result in 5 CHIP commands running in parallel, that will each create chips for 15 scenes.

The command DAG that is given to the runner is constructed such that each split command can be run in parallel if the runner supports parallelization, and that any command that is dependent on the output of the split command will be dependent on each of the splits. So that means, in the above example, a `TRAIN` command, which was dependent on a single `CHIP` command pre-split, will be dependent each of the 5 individual `CHIP` commands after the split.

Each runner will handle parallelization differently. For instance, the local runner will run each of the splits simultaneously, so be sure the split number is in relation to the number of CPUs available. The AWS Batch runner will use array jobs to run commands in parallel, and the Batch Compute Environment will determine how many resources are available to run jobs simultaneously.

CHAPTER 11

Miscellaneous Topics

11.1 File Systems

The `FileSystem` architecture supports multiple file systems through an interface that is chosen by URI. There is built-in support for: local and HTTP file systems in the `rastervision.pipeline` package, AWS S3 in the `rastervision.aws_s3` plugin, and any file that can be opened using GDAL VSI in the `rastervision.gdal_vsi` plugin. Some file systems support read only (HTTP), while others are read/write. If you need to support other file storage systems, you can add new `FileSystem` classes via a plugin.

11.2 Viewing Tensorboard

The PyTorch backends included in the `rastervision.pytorch_backend` plugin will start an instance of TensorBoard while training if `log_tensorboard=True` and `run_tensorboard=True` in the `BackendConfig`. To view TensorBoard, go to `https://<domain>:6006/`. If you're running locally, then `<domain>` should be `localhost`, and if you are running remotely (for example AWS), <domain> is the public DNS of the machine running the training command. If running locally, make sure to forward port 6006 using the `--tensorboard` option to `docker/run` if you are using it. At the moment, basic metrics are logged each epoch, but more interesting visualization could be added in the future.

11.3 Transfer learning using models trained by RV

To use a model trained by Raster Vision for transfer learning or fine tuning, you can use output of the TRAIN command of the experiment as a pretrained model of further experiments. The `last_model.pth` model file in the `train` directory can be used as a pretrained model in a new pipeline. To do so, set the `init_weights` field to the model file in the `ModelConfig` in the new pipeline.

11.4 Making Predictions with Model Bundles

To make predictions on new imagery, the *bundle* command generates a "model bundle" which can be used with the *predict* command. This loads the model and saves the predictions for a single scene. If you need to call this for a large number of scenes, consider using the `Predictor` class programmatically, as this will allow you to load the model once and use it many times. This can matter a lot if you want the time-to-prediction to be as fast as possible - the model load time can be orders of magnitudes slower than the prediction time of a loaded model.

The model bundle is a zip file containing the model weights and the configuration necessary for Raster Vision to use the model. This configuration includes the configuration of the model architecture, how the training data was processed by *RasterTransformer*, the subset of bands used by the *RasterSource*, and potentially other things. The model bundle holds all of this necessary information, so that a prediction call only needs to know what imagery it is predicting against.

This works generically over all models produced by Raster Vision, without additional client considerations, and therefore abstracts away the specifics of every model when considering how to deploy prediction software. Note that this means that by default, predictions will be made according to the configuration of the pipeline that produced the model bundle. Some of this configuration might be inappropriate for the new imagery (such as the `channel_order`), and can be overridden by options to the *predict* command.

CHAPTER 12

Contributing

We are happy to take contributions! It is best to get in touch with the maintainers about larger features or design changes *before* starting the work, as it will make the process of accepting changes smoother.

12.1 Contributor License Agreement (CLA)

Everyone who contributes code to Raster Vision will be asked to sign the Azavea CLA, which is based off of the Apache CLA.

1. Download a copy of the Raster Vision Individual Contributor License Agreement or the Raster Vision Corporate Contributor License Agreement
2. Print out the CLAs and sign them, or use PDF software that allows placement of a signature image.
3. Send the CLAs to Azavea by one of: - Scanning and emailing the document to cla@azavea.com - Faxing a copy to +1-215-925-2600. - Mailing a hardcopy to: `Azavea, 990 Spring Garden Street, 5th Floor, Philadelphia, PA 19107 USA`

CHAPTER 13

Release Process

This is a guide to the process of creating a new release, and is meant for the maintainers of Raster Vision.

Note: The following instructions assume that Python 3 is the default Python on your local system. Using Python 2 will not work.

13.1 Minor or Major Version Release

1. It's a good idea to update any major dependencies before the release.
2. Fix any broken badges on the Github repo readme.
3. Update the docs to reflect all changes since last release. You can view a local copy of the docs by running `docker/run --docs` and then viewing `localhost:8000`.
4. Update `tiny_spacenet.py` if needed and copy its contents to the index, Quickstart, and Github repo readme.
5. Update Changelog, and point out config API changes.
6. Test out *Setup* and *Quickstart* instructions and make sure they work.
7. Test examples from *Writing pipelines and plugins*, *Bootstrap new projects with a template*, and *Setup AWS Batch using CloudFormation*.

```
rastervision run inprocess rastervision.pipeline_example_plugin1.config1 -
↪a root_uri /opt/data/pipeline-example/1/ -s 2
rastervision run inprocess rastervision.pipeline_example_plugin1.config2 -
↪a root_uri /opt/data/pipeline-example/2/ -s 2
rastervision run inprocess rastervision.pipeline_example_plugin2.config3 -
↪a root_uri /opt/data/pipeline-example/3/ -s 2
```

```
cookiecutter /Users/lfishgold/projects/raster-vision/cookiecutter_template
```

8. Run all *Examples* and check that evaluation metrics are close to the scores from the last release. (For each example, there should be a link to a JSON file with the evaluation metrics from the last release.) This stage often uncovers bugs, and is the most time consuming part of the release process. There is a script to help run the examples and collect their output in `rastervision.pytorch_backend.examples.test`. There are the following subcommands: `run` to run sets of examples remotely or locally, `collect` to download certain files generated by running examples to aide inspection, and `predict` to run the predict command on example model bundles. To use this script, you will need to follow certain conventions around file organization which will be apparent in the `cfg` dictionary in the source code for `test.py`.
9. Collect all model bundles, and check that they work with the `predict` command and sanity check output in QGIS.
10. Update the *Model Zoo* by uploading model bundles and sample images to the right place on S3. If you use the `collect` command (described above), you should be able to sync the `collect_dir` to `s3://azavea-research-public-data/raster-vision/examples/model-zoo-<version>`.
11. Update the version number. This occurs in many, many places, so it's best to do this with a find and replace over the entire repo.
12. Make a PR to the `master` branch with the preceding updates. In the PR, there should be a link to preview the docs. Check that they are building and look correct.
13. Make a git branch with the version as the name, and push to Github.
14. Ensure that the docs are building correctly for the new version branch on readthedocs. You will need to have admin access on your RTD account. Once the branch is building successfully, Under Versions -> Activate a Version, you can activate the version to add it to the sidebar of the docs for the latest version. (This might require manually triggering a rebuild of the docs.) Then, under Admin -> Advanced Settings, change the default version to the new version.
15. Travis CI is supposed to publish an image whenever there is a push to a branch with a version number as the name. If this doesn't work or you want to publish it immediately, then you can manually make a Docker image for the new version and push to Quay. For this you will need an account on Quay.io under the Azavea organization.

```
./docker/build
docker login quay.io
docker tag raster-vision-pytorch:latest quay.io/azavea/raster-
↪vision:pytorch-<version>
docker push quay.io/azavea/raster-vision:pytorch-<version>
```

16. Make a Github tag and release using the previous release as a template.
17. Publish all packages to PyPI. This step requires `twine` which you can install with `pip install twine`. To store settings for PyPI you can setup a `~/.pypirc` file containing:

```
[pypi]
username = azavea
```

Once packages are published they cannot be changed so be careful. (It's possible to practice using testpypi.) Navigate to the `raster-vision` repo on your local filesystem. With the version branch checked out, run something like the following to publish each plugin, and then the top-level package.

```
export RV="/Users/lfishgold/projects/raster-vision"

cd $RV/rastervision_pipeline
```

(continues on next page)

(continued from previous page)

```
python setup.py sdist bdist_wheel
twine upload dist/*

cd $RV/rastervision_aws_batch
python setup.py sdist bdist_wheel
twine upload dist/*

cd $RV/rastervision_aws_s3
python setup.py sdist bdist_wheel
twine upload dist/*

cd $RV/rastervision_core
python setup.py sdist bdist_wheel
twine upload dist/*

cd $RV/rastervision_pytorch_learner
python setup.py sdist bdist_wheel
twine upload dist/*

cd $RV/rastervision_pytorch_backend
python setup.py sdist bdist_wheel
twine upload dist/*

cd $RV/rastervision_gdal_vsi
python setup.py sdist bdist_wheel
twine upload dist/*

cd $RV
python setup.py sdist bdist_wheel
twine upload dist/*
```

18. Announce new release on Gitter, and with blog post if it's a big release.

13.2 Bug Fix Release

This describes how to create a new bug fix release, using incrementing from 0.8.0 to 0.8.1 as an example. This assumes that there is already a branch for a minor release called `0.8`.

1. To create a bug fix release (version 0.8.1), we need to backport all the bug fix commits on the `master` branch that have been added since the last bug fix release onto the `0.8` branch. For each bug fix PR on `master`, we need to create a PR against the `0.8` branch based on a branch of `0.8` that has cherry-picked the commits from the original PR. The title of the PR should start with [BACKPORT].
2. Make and merge a PR against `0.8` (but not `master`) that increments the version in each `setup.py` file to `0.8.1`. Then wait for the `0.8` branch to be built by Travis and the `0.8` Docker images to be published to Quay. If that is successful, we can proceed to the next steps of actually publishing a release.
3. Using the Github UI, make a new release. Use `0.8.1` as the tag, and the `0.8` branch as the target.
4. The image for `0.8` is created automatically by Travis, but we need to manually create images for `0.8.1`. For this you will need an account on Quay under the Azavea organization.

```
docker login quay.io

docker pull quay.io/azavea/raster-vision:pytorch-0.8
```

(continues on next page)

(continued from previous page)

```
docker tag quay.io/azavea/raster-vision:pytorch-0.8 quay.io/azavea/raster-
↪vision:pytorch-0.8.1
docker push quay.io/azavea/raster-vision:pytorch-0.8.1
```

5. Publish the new version to PyPI. Follow the same instructions for PyPI that are listed above for minor/major version releases.

CHAPTER 14

Configuration API

14.1 Configuration API Reference

This contains the API used for configuring various components of Raster Vision pipelines. This serves as the lower-level companion to the discussion of *Pipelines and Commands*.

14.1.1 rastervision.pipeline

14.1.2 rastervision.core

StatsAnalyzerConfig

class rastervision.core.analyzer.**StatsAnalyzerConfig**

Config for an Analyzer that computes imagery statistics of scenes.

output_uri

URI for output. If None and this is part of an RVPipeline, this is auto-generated. Defaults to None.

Type Optional[str]

sample_prob

The probability of using a random window for computing statistics. If None, will use a sliding window. Defaults to 0.1.

Type Optional[float]

ClassConfig

class rastervision.core.data.**ClassConfig**

Configures the class names that are being predicted.

names

Names of classes.

Type List[str]

colors
Colors used to visualize classes. Can be color strings accepted by matplotlib or RGB tuples. If None, a random color will be auto-generated for each class. Defaults to None.

Type Optional[List[Union[List, str]]]

null_class
Optional name of class in *names* to use as the null class. This is used in semantic segmentation to represent the label for imagery pixels that are NODATA or that are missing a label. If None, and this Config is part of a SemanticSegmentationConfig, a null class will be added automatically. Defaults to None.

Type Optional[str]

DatasetConfig

class rastervision.core.data.**DatasetConfig**
Config for a Dataset comprising the scenes for train, valid, and test splits.

class_config

Type *ClassConfig*

train_scenes

Type List[*SceneConfig*]

validation_scenes

Type List[*SceneConfig*]

test_scenes
Defaults to [].

Type List[*SceneConfig*]

img_channels
The number of channels of the images. Defaults to None.

Type Optional[PositiveInt]

SceneConfig

class rastervision.core.data.**SceneConfig**
Config for a Scene which comprises the raster data and labels for an AOI.

id

Type str

raster_source

Type RasterSourceConfig

label_source

Type LabelSourceConfig

label_store
Defaults to None.

Type Optional[LabelStoreConfig]

aoi_geometries
An array of GeoJSON geometries represented as Python dictionaries. Defaults to None.

Type Optional[List[dict]]

aoi_uris
List of URIs of GeoJSON files that define the AOIs for the scene. Each polygondefines an AOI which is a piece of the scene that is assumed to be fully labeled and usable for training or validation. Defaults to None.

Type Optional[List[str]]

ChipClassificationLabelSourceConfig

class rastervision.core.data.label_source.**ChipClassificationLabelSourceConfig**
Config for a source of labels for chip classification.

This can be provided explicitly as a grid of cells, or a grid of cells can be inferred from arbitrary polygons.

vector_source

Type *VectorSourceConfig*

ioa_thresh
Minimum IOA of a polygon and cell for that polygon to be a candidate for setting the class_id. Defaults to None.

Type Optional[float]

use_intersection_over_cell
If True, then use the area of the cell as the denominator in the IOA. Otherwise, use the area of the polygon. Defaults to False.

Type bool

pick_min_class_id
If True, the class_id for a cell is the minimum class_id of the boxes in that cell. Otherwise, pick the class_id of the box covering the greatest area. Defaults to False.

Type bool

background_class_id
If not None, class_id to use as the background class; ie. the one that is used when a window contains no boxes. If not set, empty windows have None set as their class_id which is considered a null value. Defaults to None.

Type Optional[int]

infer_cells
If True, infers a grid of cells based on the cell_sz. Defaults to False.

Type bool

cell_sz
Size of a cell to use in pixels. If None, and this Config is part of an RVPipeline, this field will be set from RVPipeline.train_chip_sz. Defaults to None.

Type Optional[int]

lazy
If True, labels will not be populated automatically during initialization of the label source. Defaults to False.

Type bool

SemanticSegmentationLabelSourceConfig

class rastervision.core.data.label_source.**SemanticSegmentationLabelSourceConfig**
Config for a read-only label source for semantic segmentation.

raster_source
The labels in the form of rasters.

Type Union[None.RasterSourceConfig, None.RasterizedSourceConfig]

rgb_class_config
If set, will infer the class_ids for the labels using the colors field. This assumes the labels are stored as RGB rasters. Defaults to None.

Type Optional[*ClassConfig*]

ObjectDetectionLabelSourceConfig

class rastervision.core.data.label_source.**ObjectDetectionLabelSourceConfig**
Config for a read-only label source for object detection.

vector_source

Type *VectorSourceConfig*

ChipClassificationGeoJSONStoreConfig

class rastervision.core.data.label_store.**ChipClassificationGeoJSONStoreConfig**
Config for storage for chip classification predictions.

uri
URI of GeoJSON file with predictions. If None, and this Config is part of a SceneConfig inside an RVPipelineConfig, it will be auto-generated. Defaults to None.

Type Optional[str]

PolygonVectorOutputConfig

class rastervision.core.data.label_store.**PolygonVectorOutputConfig**
Config for vectorized semantic segmentation predictions.

uri
URI of vector output. If None, and this Config is part of a SceneConfig and RVPipeline, this field will be auto-generated. Defaults to None.

Type Optional[str]

class_id
The prediction class that is to turned into vectors.

Type int

denoise
Radius of the structural element used to remove high-frequency signals from the image. Defaults to 0.

Type int

BuildingVectorOutputConfig

class rastervision.core.data.label_store.**BuildingVectorOutputConfig**

Config for vectorized semantic segmentation predictions.

Intended to break up clusters of buildings.

uri

URI of vector output. If None, and this Config is part of a SceneConfig and RVPipeline, this field will be auto-generated. Defaults to None.

Type Optional[str]

class_id

The prediction class that is to turned into vectors.

Type int

denoise

Radius of the structural element used to remove high-frequency signals from the image. Defaults to 0.

Type int

min_aspect_ratio

Ratio between length and height (or height and length) of anything that can be considered to be a cluster of buildings. The goal is to distinguish between rows of buildings and (say) a single building. Defaults to 1.618.

Type float

min_area

Minimum area of anything that can be considered to be a cluster of buildings. The goal is to distinguish between buildings and artifacts. Defaults to 0.0.

Type float

element_width_factor

Width of the structural element used to break building clusters as a fraction of the width of the cluster. Defaults to 0.5.

Type float

element_thickness

Thickness of the structural element that is used to break building clusters. Defaults to 0.001.

Type float

SemanticSegmentationLabelStoreConfig

class rastervision.core.data.label_store.**SemanticSegmentationLabelStoreConfig**

Config for storage for semantic segmentation predictions.

Stores class raster as GeoTIFF, and can optionally vectorizes predictions and stores them in GeoJSON files.

uri

URI of file with predictions. If None, and this Config is part of a SceneConfig inside an RVPipelineConfig, this fiend will be auto-generated. Defaults to None.

Type Optional[str]

vector_output

Defaults to [].

Type List[VectorOutputConfig]

rgb
If True, save prediction class_ids in RGB format using the colors in class_config. Defaults to False.

Type bool

smooth_output
If True, expects labels to be continuous values representing class scores and stores both scores and discrete labels. Defaults to False.

Type bool

smooth_as_uint8
If True, stores smooth scores as uint8, resulting in loss of precision, but reduced file size. Only used if smooth_output=True. Defaults to False.

Type bool

rasterio_block_size
blockxsize and blockysize params in rasterio.open() will be set to this. Defaults to 256.

Type int

ObjectDetectionGeoJSONStoreConfig

class rastervision.core.data.label_store.**ObjectDetectionGeoJSONStoreConfig**
Config for storage for object detection predictions.

uri
URI of GeoJSON file with predictions. If None, and this Config is part of a SceneConfig inside an RVPipelineConfig, it will be auto-generated. Defaults to None.

Type Optional[str]

RasterioSourceConfig

class rastervision.core.data.raster_source.**RasterioSourceConfig**

channel_order
The sequence of channel indices to use when reading imagery. Defaults to None.

Type Optional[List[int]]

transformers
Defaults to [].

Type List[RasterTransformerConfig]

extent_crop
Relative offsets (skip_top, skip_left, skip_bottom, skip_right) for cropping the extent of the raster source. Useful for splitting a scene into different dataset splits. E.g. if you want to use the top 80% of the image for training and the bottom 20% for validation you can pass extent_crop=CropOffsets(skip_bottom=0.20) to the raster source in the training scene and extent_crop=CropOffsets(skip_top=0.80) to the raster source in the validation scene. Defaults to None i.e. no cropping. Defaults to None.

Type Optional[CropOffsets]

uris
List of image URIs that comprise imagery for a scene. The format of each file can be any that can be read by Rasterio/GDAL. If > 1 URI is provided, a VRT will be created to mosaic together the individual images.

Type List[str]

allow_streaming
Allow streaming of assets rather than always downloading. Defaults to False.

Type bool

x_shift
Defaults to 0.0.

Type float

y_shift
Defaults to 0.0.

Type float

RasterizerConfig

class rastervision.core.data.raster_source.**RasterizerConfig**

background_class_id
The class_id to use for any background pixels, ie. pixels not covered by a polygon.

Type int

all_touched
If True, all pixels touched by geometries will be burned in. If false, only pixels whose center is within the polygon or that are selected by Bresenham's line algorithm will be burned in. (See rasterio.features.rasterize). Defaults to False.

Type bool

RasterizedSourceConfig

class rastervision.core.data.raster_source.**RasterizedSourceConfig**

vector_source

Type *VectorSourceConfig*

rasterizer_config

Type *RasterizerConfig*

MultiRasterSourceConfig

class rastervision.core.data.raster_source.**MultiRasterSourceConfig**

channel_order
The sequence of channel indices to use when reading imagery. Defaults to None.

Type Optional[List[int]]

transformers
Defaults to [].

Type List[RasterTransformerConfig]

extent_crop
Relative offsets (skip_top, skip_left, skip_bottom, skip_right) for cropping the extent of the raster source. Useful for splitting a scene into different dataset splits. E.g. if you want to use the top 80% of the image for training and the bottom 20% for validation you can pass extent_crop=CropOffsets(skip_bottom=0.20) to the raster source in the training scene and extent_crop=CropOffsets(skip_top=0.80) to the raster source in the validation scene. Defaults to None i.e. no cropping. Defaults to None.

Type Optional[CropOffsets]

raster_sources
List of SubRasterSourceConfigs to combine.

Type Sequence[SubRasterSourceConfig]

allow_different_extents
Allow sub-rasters to have different extents. Defaults to False.

Type bool

force_same_dtype
Force all subchips to be of the same dtype as the first subchip. Defaults to False.

Type bool

crs_source
Use the crs_transformer of the raster source at this index. Defaults to 0.

Type ConstrainedIntValue

StatsTransformerConfig

class rastervision.core.data.raster_transformer.**StatsTransformerConfig**

stats_uri
The URI of the output of the StatsAnalyzer. If None, and this Config is inside an RVPipeline, then this field will be auto-generated. Defaults to None.

Type Optional[str]

CastTransformerConfig

class rastervision.core.data.raster_transformer.**CastTransformerConfig**

to_dtype
dtype to cast raster to. Must be a valid Numpy dtype e.g. “uint8”, “float32”, etc.

Type str

NanTransformerConfig

class rastervision.core.data.raster_transformer.**NanTransformerConfig**

to_value
Turn all NaN values into this value. Defaults to 0.0.

Type Optional[float]

ReclassTransformer

class rastervision.core.data.raster_transformer.**ReclassTransformer**(*mapping: Dict[int, int]*)

Reclassifies label raster

VectorSourceConfig

class rastervision.core.data.vector_source.**VectorSourceConfig**

default_class_id
The default class_id to use if class cannot be inferred using other mechanisms. If a feature has an inferred class_id of None, then it will be deleted.

Type Optional[int]

class_id_to_filter
Map from class_id to JSON filter used to infer missing class_ids. Each key should be a class id, and its value should be a boolean expression which is run against the property field for each feature.This allows matching different features to different class ids based on its properties. The expression schema is that described by https://docs.mapbox.com/mapbox-gl-js/style-spec/other/#other-filter. Defaults to None.

Type Optional[Dict]

line_bufs
This is useful, for example, for buffering lines representing roads so that their width roughly matches the width of roads in the imagery. If None, uses default buffer value of 1. Otherwise, a map from class_id to number of pixels to buffer by. If the buffer value is None, then no buffering will be performed and the LineString or Point won't get converted to a Polygon. Not converting to Polygon is incompatible with the currently available LabelSources, but may be useful in the future. Defaults to None.

Type Optional[Mapping[int, Union[int, float, NoneType]]]

point_bufs
Same as above, but used for buffering Points into Polygons. Defaults to None.

Type Optional[Mapping[int, Union[int, float, NoneType]]]

GeoJSONVectorSourceConfig

class rastervision.core.data.vector_source.**GeoJSONVectorSourceConfig**

default_class_id
The default class_id to use if class cannot be inferred using other mechanisms. If a feature has an inferred class_id of None, then it will be deleted.

Type Optional[int]

class_id_to_filter
Map from class_id to JSON filter used to infer missing class_ids. Each key should be a class id, and its value should be a boolean expression which is run against the property field for each feature.This allows matching different features to different class ids based on its properties. The expression schema is that described by https://docs.mapbox.com/mapbox-gl-js/style-spec/other/#other-filter. Defaults to None.

Type Optional[Dict]

line_bufs
This is useful, for example, for buffering lines representing roads so that their width roughly matches the width of roads in the imagery. If None, uses default buffer value of 1. Otherwise, a map from class_id to number of pixels to buffer by. If the buffer value is None, then no buffering will be performed and the LineString or Point won't get converted to a Polygon. Not converting to Polygon is incompatible with the currently available LabelSources, but may be useful in the future. Defaults to None.

Type Optional[Mapping[int, Union[int, float, NoneType]]]

point_bufs
Same as above, but used for buffering Points into Polygons. Defaults to None.

Type Optional[Mapping[int, Union[int, float, NoneType]]]

uri
The URI of a GeoJSON file.

Type str

ignore_crs_field
Defaults to False.

Type bool

ChipClassificationEvaluatorConfig

class rastervision.core.evaluation.**ChipClassificationEvaluatorConfig**

output_uri
URI of JSON output by evaluator. If None, and this Config is part of an RVPipeline, then this field will be auto-generated. Defaults to None.

Type Optional[str]

SemanticSegmentationEvaluatorConfig

class rastervision.core.evaluation.**SemanticSegmentationEvaluatorConfig**

output_uri
URI of JSON output by evaluator. If None, and this Config is part of an RVPipeline, then this field will be auto-generated. Defaults to None.

Type Optional[str]

vector_output_uri
URI of evaluation of vector output. If None, and this Config is part of an RVPipeline, then this field will be auto-generated. Defaults to None.

Type Optional[str]

ObjectDetectionEvaluatorConfig

class rastervision.core.evaluation.**ObjectDetectionEvaluatorConfig**

output_uri
URI of JSON output by evaluator. If None, and this Config is part of an RVPipeline, then this field will be auto-generated. Defaults to None.

Type Optional[str]

ChipClassificationConfig

class rastervision.core.rv_pipeline.**ChipClassificationConfig**

root_uri
The root URI for output generated by the pipeline. Defaults to None.

Type Optional[str]

rv_config
Used to store serialized RVConfig so pipeline can run in remote environment with the local RVConfig. This should not be set explicitly by users – it is only used by the runner when running a remote pipeline. Defaults to None.

Type Optional[dict]

plugin_versions
Used to store a mapping of plugin module paths to the latest version number. This should not be set explicitly by users – it is set automatically when serializing and saving the config to disk. Defaults to None.

Type Optional[Mapping[str, int]]

dataset
Dataset containing train, validation, and optional test scenes.

Type *DatasetConfig*

backend
Backend to use for interfacing with ML library.

Type BackendConfig

evaluators
Evaluators to run during analyzer command. If list is empty the default evaluator is added. Defaults to [].

Type List[EvaluatorConfig]

analyzers
Analyzers to run during analyzer command. A StatsAnalyzer will be added automatically if any scenes have a RasterTransformer. Defaults to [].

Type List[AnalyzerConfig]

train_chip_sz
Size of training chips in pixels. Defaults to 300.

Type int

predict_chip_sz
Size of predictions chips in pixels. Defaults to 300.

Type int

predict_batch_sz
Batch size to use during prediction. Defaults to 8.

Type int

chip_nodata_threshold
Discard chips where the proportion of NODATA values is greater than or equal to this value. Might result in false positives if there are many legitimate black pixels in the chip. Use with caution. Defaults to 1.

Type ConstrainedFloatValue

analyze_uri
URI for output of analyze. If None, will be auto-generated. Defaults to None.

Type Optional[str]

chip_uri
URI for output of chip. If None, will be auto-generated. Defaults to None.

Type Optional[str]

train_uri
URI for output of train. If None, will be auto-generated. Defaults to None.

Type Optional[str]

predict_uri
URI for output of predict. If None, will be auto-generated. Defaults to None.

Type Optional[str]

eval_uri
URI for output of eval. If None, will be auto-generated. Defaults to None.

Type Optional[str]

bundle_uri
URI for output of bundle. If None, will be auto-generated. Defaults to None.

Type Optional[str]

source_bundle_uri
If provided, the model will be loaded from this bundle for the train stage. Useful for fine-tuning. Defaults to None.

Type Optional[str]

SemanticSegmentationWindowMethod

class rastervision.core.rv_pipeline.**SemanticSegmentationWindowMethod**
Enum for window methods

sliding
use a sliding window

random_sample
randomly sample windows

SemanticSegmentationChipOptions

class rastervision.core.rv_pipeline.**SemanticSegmentationChipOptions**

Chipping options for semantic segmentation.

window_method

Window method to use for chipping. Defaults to <SemanticSegmentationWindowMethod.sliding: 'sliding'>.

Type enum

target_class_ids

List of class ids considered as targets (ie. those to prioritize when creating chips) which is only used in conjunction with the target_count_threshold and negative_survival_probability options. Applies to the random_sample window method. Defaults to None.

Type Optional[List[int]]

negative_survival_prob

List of class ids considered as targets (ie. those to prioritize when creating chips) which is only used in conjunction with the target_count_threshold and negative_survival_probability options. Applies to the random_sample window method. Defaults to 1.0.

Type float

chips_per_scene

Number of chips to generate per scene. Applies to the random_sample window method. Defaults to 1000.

Type int

target_count_threshold

Minimum number of pixels covering target_classes that a chip must have. Applies to the random_sample window method. Defaults to 1000.

Type int

stride

Stride of windows across image. Defaults to half the chip size. Applies to the sliding_window method. Defaults to None.

Type Optional[int]

SemanticSegmentationConfig

class rastervision.core.rv_pipeline.**SemanticSegmentationConfig**

root_uri

The root URI for output generated by the pipeline. Defaults to None.

Type Optional[str]

rv_config

Used to store serialized RVConfig so pipeline can run in remote environment with the local RVConfig. This should not be set explicitly by users – it is only used by the runner when running a remote pipeline. Defaults to None.

Type Optional[dict]

plugin_versions

Used to store a mapping of plugin module paths to the latest version number. This should not be set

explicitly by users – it is set automatically when serializing and saving the config to disk. Defaults to None.

Type Optional[Mapping[str, int]]

dataset
Dataset containing train, validation, and optional test scenes.

Type *DatasetConfig*

backend
Backend to use for interfacing with ML library.

Type BackendConfig

evaluators
Evaluators to run during analyzer command. If list is empty the default evaluator is added. Defaults to [].

Type List[EvaluatorConfig]

analyzers
Analyzers to run during analyzer command. A StatsAnalyzer will be added automatically if any scenes have a RasterTransformer. Defaults to [].

Type List[AnalyzerConfig]

train_chip_sz
Size of training chips in pixels. Defaults to 300.

Type int

predict_chip_sz
Size of predictions chips in pixels. Defaults to 300.

Type int

predict_batch_sz
Batch size to use during prediction. Defaults to 8.

Type int

chip_nodata_threshold
Discard chips where the proportion of NODATA values is greater than or equal to this value. Might result in false positives if there are many legitimate black pixels in the chip. Use with caution. Defaults to 1.

Type ConstrainedFloatValue

analyze_uri
URI for output of analyze. If None, will be auto-generated. Defaults to None.

Type Optional[str]

chip_uri
URI for output of chip. If None, will be auto-generated. Defaults to None.

Type Optional[str]

train_uri
URI for output of train. If None, will be auto-generated. Defaults to None.

Type Optional[str]

predict_uri
URI for output of predict. If None, will be auto-generated. Defaults to None.

Type Optional[str]

eval_uri
URI for output of eval. If None, will be auto-generated. Defaults to None.

Type Optional[str]

bundle_uri
URI for output of bundle. If None, will be auto-generated. Defaults to None.

Type Optional[str]

source_bundle_uri
If provided, the model will be loaded from this bundle for the train stage. Useful for fine-tuning. Defaults to None.

Type Optional[str]

chip_options
Defaults to SemanticSegmentationChipOptions(window_method=<SemanticSegmentationWindowMethod.sliding: 'sliding'>, target_class_ids=None, negative_survival_prob=1.0, chips_per_scene=1000, target_count_threshold=1000, stride=None, type_hint='semantic_segmentation_chip_options').

Type *SemanticSegmentationChipOptions*

predict_options
Defaults to SemanticSegmentationPredictOptions(type_hint='semantic_segmentation_predict_options', stride=None).

Type SemanticSegmentationPredictOptions

channel_display_groups
Groups of image channels to display together as a subplot when plotting the data and predictions. Can be a list or tuple of groups (e.g. [(0, 1, 2), (3,)]) or a dict containing title-to-group mappings (e.g. {"RGB": [0, 1, 2], "IR": [3]}), where each group is a list or tuple of channel indices and title is a string that will be used as the title of the subplot for that group. Defaults to None.

Type Union[dict, list, tuple, NoneType]

img_format
The filetype of the training images. Defaults to None.

Type Optional[str]

label_format
The filetype of the training labels. Defaults to 'png'.

Type str

ObjectDetectionWindowMethod

class rastervision.core.rv_pipeline.**ObjectDetectionWindowMethod**
Enum for window methods

chip
the default method

ObjectDetectionChipOptions

class rastervision.core.rv_pipeline.**ObjectDetectionChipOptions**

neg_ratio
The ratio of negative chips (those containing no bounding boxes) to positive chips. This can be useful if the statistics of the background is different in positive chips. For example, in car detection, the positive chips will always contain roads, but no examples of rooftops since cars tend to not be near rooftops. Defaults to 1.0.

Type float

ioa_thresh
When a box is partially outside of a training chip, it is not clear if (a clipped version) of the box should be included in the chip. If the IOA (intersection over area) of the box with the chip is greater than ioa_thresh, it is included in the chip. Defaults to 0.8.

Type float

window_method
Defaults to <ObjectDetectionWindowMethod.chip: 'chip'>.

Type enum

label_buffer
Defaults to None.

Type Optional[int]

ObjectDetectionPredictOptions

class rastervision.core.rv_pipeline.**ObjectDetectionPredictOptions**

merge_thresh
If predicted boxes have an IOA (intersection over area) greater than merge_thresh, then they are merged into a single box during postprocessing. This is needed since the sliding window approach results in some false duplicates. Defaults to 0.5.

Type float

score_thresh
Predicted boxes are only output if their score is above score_thresh. Defaults to 0.5.

Type float

ObjectDetectionConfig

class rastervision.core.rv_pipeline.**ObjectDetectionConfig**

root_uri
The root URI for output generated by the pipeline. Defaults to None.

Type Optional[str]

rv_config
Used to store serialized RVConfig so pipeline can run in remote environment with the local RVConfig. This should not be set explicitly by users – it is only used by the runner when running a remote pipeline. Defaults to None.

Type Optional[dict]

plugin_versions
Used to store a mapping of plugin module paths to the latest version number. This should not be set explicitly by users – it is set automatically when serializing and saving the config to disk. Defaults to None.

Type Optional[Mapping[str, int]]

dataset
Dataset containing train, validation, and optional test scenes.

Type *DatasetConfig*

backend
Backend to use for interfacing with ML library.

Type BackendConfig

evaluators
Evaluators to run during analyzer command. If list is empty the default evaluator is added. Defaults to [].

Type List[EvaluatorConfig]

analyzers
Analyzers to run during analyzer command. A StatsAnalyzer will be added automatically if any scenes have a RasterTransformer. Defaults to [].

Type List[AnalyzerConfig]

train_chip_sz
Size of training chips in pixels. Defaults to 300.

Type int

predict_chip_sz
Size of predictions chips in pixels. Defaults to 300.

Type int

predict_batch_sz
Batch size to use during prediction. Defaults to 8.

Type int

chip_nodata_threshold
Discard chips where the proportion of NODATA values is greater than or equal to this value. Might result in false positives if there are many legitimate black pixels in the chip. Use with caution. Defaults to 1.

Type ConstrainedFloatValue

analyze_uri
URI for output of analyze. If None, will be auto-generated. Defaults to None.

Type Optional[str]

chip_uri
URI for output of chip. If None, will be auto-generated. Defaults to None.

Type Optional[str]

train_uri
URI for output of train. If None, will be auto-generated. Defaults to None.

Type Optional[str]

predict_uri
URI for output of predict. If None, will be auto-generated. Defaults to None.

Type Optional[str]

eval_uri
URI for output of eval. If None, will be auto-generated. Defaults to None.

Type Optional[str]

bundle_uri
URI for output of bundle. If None, will be auto-generated. Defaults to None.

Type Optional[str]

source_bundle_uri
If provided, the model will be loaded from this bundle for the train stage. Useful for fine-tuning. Defaults to None.

Type Optional[str]

chip_options
Defaults to ObjectDetectionChipOptions(neg_ratio=1.0, ioa_thresh=0.8, window_method=<ObjectDetectionWindowMethod.chip: 'chip'>, label_buffer=None, type_hint='object_detection_chip_options').

Type *ObjectDetectionChipOptions*

predict_options
Defaults to ObjectDetectionPredictOptions(type_hint='object_detection_predict_options', merge_thresh=0.5, score_thresh=0.5).

Type *ObjectDetectionPredictOptions*

14.1.3 rastervision.pytorch_backend

PyTorchChipClassificationConfig

PyTorchSemanticSegmentationConfig

PyTorchObjectDetectionConfig

14.1.4 rastervision.pytorch_learner

Backbone

SolverConfig

ExternalModuleConfig

DataConfig

ImageDataConfig

GeoDataConfig

GeoDataWindowConfig

PlotOptions

CHAPTER 15

CHANGELOG

15.1 CHANGELOG

15.1.1 Raster Vision 0.13

This release presents a major jump in Raster Vision's power and flexibility. The most significant changes are:

Support arbitrary models and loss functions (#985, #992)

Raster Vision is no longer restricted to using the built in models and loss functions. It is now possible to import models and loss functions from a GitHub repo or a URI or a zip file as long as they interface correctly with RV's learner code. This means that you can now easily swap models in your existing training pipelines, allowing you to take advantage of the latest models or to make customizations that help with your specific task; all with minimal changes.

This is made possible by PyTorch's `hub` module.

Currently not supported for Object Detection.

Support for multiband images (even with Transfer Learning) (#972)

It is now possible to train on imagery with more than 3 channels. Raster Vision automatically modifies the model to be able to accept more than 3 channels. If using pretrained models, the pre-learned weights are retained.

The model modification cannot be performed automatically when using an external model. But as long as the external model supports multiband inputs, it will work correctly with RV.

Currently only supported for Semantic Segmentation.

Support for reading directly from raster sources during training without chipping (#1046)

It is no longer necessary to go through a `chip` stage to produce a training dataset. You can instead provide the `DatasetConfig` directly to the PyTorch backend and RV will sample training chips on the fly during training. All

the examples now use this as the default. Check them out to see how to use this feature.

Support for arbitrary Albumentations transforms (#1001)

It is now possible to supply an arbitrarily complicated Albumentations transform for data augmentation. In the `DataConfig` subclasses, you can specify a `base_transform` that is applied every time (i.e. in training, validation, and prediction), an `aug_transform` that is only applied during training, and a `plot_transform` (via `PlotOptions`) to ensure that sample images are plotted correctly (e.g. use `plot_transform` to rescale a normalized image to 0-1).

Allow streaming reads from Rasterio sources (#1020)

It is now possible to stream chips from a remote `RasterioSource` without first downloading the entire file. To enable, set `allow_streaming=True` in the `RasterioSourceConfig`.

Analyze stage no longer necessary when using non-uint8 rasters (#972)

It is no longer necessary to go through an `analyze` stage to be able to convert non-`uint8` rasters to `uint8` chips. Chips can now be stored as `numpy` arrays, and will be normalized to `float` during training/prediction based on their specific data type. See `spacenet_vegas.py` for example usage.

Currently only supported for Semantic Segmentation.

Features

- Add support for multiband images #972
- Add support for vector output to predict command #980
- Add support for weighted loss for classification and semantic segmentation #977
- Add multi raster source #978
- Add support for fetching and saving external model definitions #985
- Add support for external loss definitions #992
- Upgrade to pyproj 2.6 #1000
- Add support for arbitrary albumentations transforms #1001
- Minor tweaks to regression learner #1013
- Add ability to specify number of PyTorch reader processes #1008
- Make img_sz specifiable #1012
- Add ignore_last_class capability to segmentation #1017
- Add filtering capability to segmentation sliding window chip generation #1018
- Add raster transformer to remove NaNs from float rasters, add raster transformers to cast to arbitrary numpy types #1016
- Add plot options for regression #1023
- Add ability to use fewer channels w/ pretrained models #1026
- Remove 4GB file size limit from VSI file system, allow streaming reads #1020

- Add reclassification transformer for segmentation label rasters #1024
- Allow filtering out chips based on proportion of NODATA pixels #1025
- Allow ignore_last_class to take either a boolean or the literal 'force'; in the latter case validation of that argument is skipped so that it can be used with external loss functions #1027
- Add ability to crop raster source extent #1030
- Accept immediate geometries in SceneConfig #1033
- Only perform normalization on unsigned integer types #1028
- Make group_uris specifiable and add group_train_sz_rel #1035
- Make number of training and dataloader previews independent of batch size #1038
- Allow continuing training from a model bundle #1022
- Allow reading directly from raster source during training without chipping #1046
- Remove external commands (obsoleted by external architectures and loss functions) #1047
- Allow saving SS predictions as probabilities #1057
- Update CUDA version from 10.1 to 10.2 #1115
- Add integration tests for the nochip functionality #1116
- Update examples to make use of the nochip functionality by default #1116

Bug Fixes

- Update all relevant saved URIs in config before instantiating Pipeline #993
- Pass verbose flag to batch jobs #988
- Fix: Ensure Integer class_id #990
- Use `--ipc=host` by default when running the docker container #1077

15.1.2 Raster Vision 0.12

This release presents a major refactoring of Raster Vision intended to simplify the codebase, and make it more flexible and customizable.

To learn about how to upgrade existing experiment configurations, perhaps the best approach is to read the source code of the *Examples* to get a feel for the new syntax. Unfortunately, existing predict packages will not be usable with this release, and upgrading and re-running the experiments will be necessary. For more advanced users who have written plugins or custom commands, the internals have changed substantially, and we recommend reading *Architecture and Customization*.

Since the changes in this release are sweeping, it is difficult to enumerate a list of all changes and associated PRs. Therefore, this change log describes the changes at a high level, along with some justifications and pointers to further documentation.

Simplified Configuration Schema

We are still using a modular, programmatic approach to configuration, but have switched to using a `Config` base class which uses the Pydantic library. This allows us to define configuration schemas in a declarative fashion, and let the underlying library handle serialization, deserialization, and validation. In addition, this has allowed us to DRY up the configuration code, eliminate the use of Protobufs, and represent configuration from plugins in the same fashion as built-in functionality. To see the difference, compare the configuration code for `ChipClassificationLabelSource` in 0.11 (label_source.proto and chip_classification_label_source_config.py), and in 0.12 (chip_classification_label_source_config.py).

Abstracted out Pipelines

Raster Vision includes functionality for running computational pipelines in local and remote environments, but previously, this functionality was tightly coupled with the "domain logic" of machine learning on geospatial data in the `Experiment` abstraction. This made it more difficult to add and modify commands, as well as use this functionality in other projects. In this release, we factored out the experiment running code into a separate *rastervision.pipeline* package, which can be used for defining, configuring, customizing, and running arbitrary computational pipelines.

Reorganization into Plugins

The rest of Raster Vision is now written as a set of optional plugins that have `Pipelines` which implement the "domain logic" of machine learning on geospatial data. Implementing everything as optional (`pip` installable) plugins makes it easier to install subsets of Raster Vision functionality, eliminates separate code paths for built-in and plugin functionality, and provides (de facto) examples of how to write plugins. See *Codebase Overview* for more details.

More Flexible PyTorch Backends

The 0.10 release added PyTorch backends for chip classification, semantic segmentation, and object detection. In this release, we abstracted out the common code for training models into a flexible `Learner` base class with subclasses for each of the computer vision tasks. This code is in the `rastervision.pytorch_learner` plugin, and is used by the `Backends` in `rastervision.pytorch_backend`. By decoupling `Backends` and `Learners`, it is now easier to write arbitrary `Pipelines` and new `Backends` that reuse the core model training code, which can be customized by overriding methods such as `build_model`. See *Customizing Raster Vision*.

Removed Tensorflow Backends

The Tensorflow backends and associated Docker images have been removed. It is too difficult to maintain backends for multiple deep learning frameworks, and PyTorch has worked well for us. Of course, it's still possible to write `Backend` plugins using any framework.

Other Changes

- For simplicity, we moved the contents of the raster-vision-examples and raster-vision-aws repos into the main repo. See *Examples* and *Setup AWS Batch using CloudFormation*.
- To help people bootstrap new projects using RV, we added *Bootstrap new projects with a template*.
- All the PyTorch backends now offer data augmentation using albumentations.
- We removed the ability to automatically skip running commands that already have output, "tree workflows", and "default providers". We also unified the `Experiment`, `Command`, and `Task` classes into a single `Pipeline`

class which is subclassed for different computer vision (or other) tasks. These features and concepts had little utility in our experience, and presented stumbling blocks to outside contributors and plugin writers.

- Although it's still possible to add new `VectorSources` and other classes for reading data, our philosophy going forward is to prefer writing pre-processing scripts to get data into the format that Raster Vision can already consume. The `VectorTileVectorSource` was removed since it violates this new philosophy.
- We previously attempted to make predictions for semantic segmentation work in a streaming fashion (to avoid running out of RAM), but the implementation was buggy and complex. So we reverted to holding all predictions for a scene in RAM, and now assume that scenes are roughly < 20,000 x 20,000 pixels. This works better anyway from a parallelization standponit.
- We switched to writing chips to disk incrementally during the `CHIP` command using a `SampleWriter` class to avoid running out of RAM.
- The term "predict package" has been replaced with "model bundle", since it rolls off the tongue better, and `BUNDLE` is the name of the command that produces it.
- Class ids are now indexed starting at 0 instead of 1, which seems more intuitive. The "null class", used for marking pixels in semantic segmentation that have not been labeled, used to be 0, and is now equal to `len(class_ids)`.
- The `aws_batch` runner was renamed `batch` due to a naming conflict, and the names of the configuration variables for Batch changed. See *Setting up AWS Batch*.

Future Work

The next big features we plan on developing are:

- the ability to read and write data in STAC format using the label extension. This will facilitate integration with other tools such as GroundWork.

15.1.3 Raster Vision 0.11

Features

- Added the possibility for chip classification to use data augmentors from the albumentations libary to enhance the training data. #859
- Updated the Quickstart doc with pytorch docker image and model #863
- Added the possibility to deal with class imbalances through oversampling. #868

Raster Vision 0.11.0

Bug Fixes

- Ensure randint args are ints #849
- The augmentors were not serialized properly for the chip command #857
- Fix problems with pretrained flag #860
- Correctly get_local_path for some zxy tile URIS #865

15.1.4 Raster Vision 0.10

Raster Vision 0.10.0

Notes on switching to PyTorch-based backends

The current backends based on Tensorflow have several problems:

- They depend on third party libraries (Deeplab, TF Object Detection API) that are complex, not well suited to being used as dependencies within a larger project, and are each written in a different style. This makes the code for each backend very different from one other, and unnecessarily complex. This increases the maintenance burden, makes it difficult to customize, and makes it more difficult to implement a consistent set of functionality between the backends.
- Tensorflow, in the maintainer's opinion, is more difficult to write and debug than PyTorch (although this is starting to improve).
- The third party libraries assume that training images are stored as PNG or JPG files. This limits our ability to handle more than three bands and more that 8-bits per channel. We have recently completed some research on how to train models on > 3 bands, and we plan on adding this functionality to Raster Vision.

Therefore, we are in the process of sunsetting the Tensorflow backends (which will probably be removed) and have implemented replacement PyTorch-based backends. The main things to be aware of in upgrading to this version of Raster Vision are as follows:

- Instead of there being CPU and GPU Docker images (based on Tensorflow), there are now tf-cpu, tf-gpu, and pytorch (which works on both CPU and GPU) images. Using `./docker/build --tf` or `./docker/build --pytorch` will only build the TF or PyTorch images, respectively.
- Using the TF backends requires being in the TF container, and similar for PyTorch. There are now `--tf-cpu`, `--tf-gpu`, and `--pytorch-gpu` options for the `./docker/run` command. The default setting is to use the PyTorch image in the standard (CPU) Docker runtime.
- The raster-vision-aws CloudFormation setup creates Batch resources for TF-CPU, TF-GPU, and PyTorch. It also now uses default AMIs provided by AWS, simplifying the setup process.
- To easily switch between running TF and PyTorch jobs on Batch, we recommend creating two separate Raster Vision profiles with the Batch resources for each of them.
- The way to use the `ConfigBuilders` for the new backends can be seen in the examples repo and the *Backend* reference

Features

- Add confusion matrix as metric for semantic segmentation #788
- Add predict_chip_size as option for semantic segmentation #786
- Handle "ignore" class for semantic segmentation #783
- Add stochastic gradient descent ("SGD") as an optimizer option for chip classification #792
- Add option to determine if all touched pixels should be rasterized for rasterized RasterSource #803
- Script to generate GeoTIFF from ZXY tile server #811
- Remove QGIS plugin #818
- Add PyTorch backends and add PyTorch Docker image #821 and #823.

Bug Fixes

- Fixed issue with configuration not being able to read lists #784
- Fixed ConfigBuilders not supporting type annotations in __init__ #800

15.1.5 Raster Vision 0.9

Raster Vision 0.9.0

Features

- Add requester_pays RV config option #762
- Unify Docker scripts #743
- Switch default branch to master #726
- Merge GeoTiffSource and ImageSource into RasterioSource #723
- Simplify/clarify/test/validate RasterSource #721
- Simplify and generalize geom processing #711
- Predict zero for nodata pixels on semantic segmentation #701
- Add support for evaluating vector output with AOIs #698
- Conserve disk space when dealing with raster files #692
- Optimize StatsAnalyzer #690
- Include per-scene eval metrics #641
- Make and save predictions and do eval chip-by-chip #635
- Decrease semseg memory usage #630
- Add support for vector tiles in .mbtiles files #601
- Add support for getting labels from zxy vector tiles #532
- Remove custom `__deepcopy__` implementation from `ConfigBuilder`s. #567
- Add ability to shift raster images by given numbers of meters. #573
- Add ability to generate GeoJSON segmentation predictions. #575
- Add ability to run the DeepLab eval script. #653
- Submit CPU-only stages to a CPU queue on Aws. #668
- Parallelize CHIP and PREDICT commands #671
- Refactor `update_for_command` to split out the IO reporting into `report_io`. #671
- Add Multi-GPU Support to DeepLab Backend #590
- Handle multiple AOI URIs #617
- Give `train_restart_dir` Default Value #626
- Use `make` to manage local execution #664
- Optimize vector tile processing #676

Bug Fixes

- Fix Deeplab resume bug: update path in checkpoint file #756
- Allow Spaces in `--channel-order` Argument #731
- Fix error when using predict packages with AOIs #674
- Correct checkpoint name #624
- Allow using default stride for semseg sliding window #745
- Fix filter_by_aoi for ObjectDetectionLabels #746
- Load null channel_order correctly #733
- Handle Rasterio crs that doesn't contain EPSG #725
- Fixed issue with saving semseg predictions for non-georeferenced imagery #708
- Fixed issue with handling width > height in semseg eval #627
- Fixed issue with experiment configs not setting key names correctly #576
- Fixed issue with Raster Sources that have channel order #576

15.1.6 Raster Vision 0.8

Raster Vision 0.8.1

Bug Fixes

- Allow multiploygon for chip classification #523
- Remove unused args for AWS Batch runner #503
- Skip over lines when doing chip classification, Use background_class_id for scenes with no polygons #507
- Fix issue where `get_matching_s3_keys` fails when `suffix` is `None` #497

Index

D

E

F

G

I

R

S

W

X

Y